The Dissertation Committee for Sankaranarayanan Gurumurthy certifies that this is the approved version of the following dissertation:

Automatic generation of instruction sequences for software-based self-test of processors and systems-on-a-chip

Committee:

Jacob A. Abraham, Supervisor

Lorenzo Alvisi

Adnan Aziz

Srinivas Patil

Nur Touba

Automatic generation of instruction sequences for software-based self-test of processors and systems-on-a-chip

by

Sankaranarayanan Gurumurthy, B.Tech, M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2008

Dedicated to my parents

Acknowledgments

As I come to the end of my journey towards a doctorate, I look back and see that there are many people without whose support and encouragement this journey would not have been a success. Without the help from these people, my pursuit for PhD might have come to an abrupt halt long time ago or might not have even started.

First and foremost, I would like to thank my advisor Prof. Jacob A. Abraham. His positive attitude and infectious enthusiasm turned the cynic in me into a believer. He has always been supportive of my endeavors during my stay here in CERC. He gave me the independence to do things my way, which has given me a lot of confidence. His insistence on looking at the big picture and the practical applications of any research that we do has stood me in good stead. He has always been there with the right suggestion whenever I hit a roadblock.

I would like to express my gratitude to Prof. Lorenzo Alvisi, Prof. Adnan Aziz, Dr. Srinivas Patil and Prof. Nur Touba for their patience and guidance while being members of my dissertation committee. The suggestions that they made during my proposal and defense were very insightful.

I am thankful to Prof. Fabio Somenzi for introducing me to research. His kind words gave me the initial confidence to pursue PhD. He is one of the

most patient people I have known. I will never forget the long discussions we used to have when he guided my Masters thesis effort.

I would like to thank my co-authors in various papers Sriram Sambamurthy, Ramtilak Vemu and Shobha Vasudevan for the fruitful discussions we had. They, along with Ramyanshu Datta, Jyotirmoy Deshmukh, Roopsha Samanta, Chaoming Zhang, Rajeshwary Tayade, Hongjoong Shin, Joonsung Park, Whitney Townsend, Adam Tate and other members of CERC, made my tenure in CERC a pleasurable one.

I thank Andrew Kieschnick for putting up with my arcane software requests and runaway experiments. I express my gratitude to Debi Prather, Melissa Campos and Linda Frost who made my stay in CERC a hassle-free one.

Li Chen, Vivek Vedula, Suriya Natarajan, Kamal Jayaraman, Rajesh Galivanche and Praveen Parvathala of Intel provided me with key insights that helped me in my research. Their suggestions helped me get an industrial perspective on my research. I would also like to thank the funding agencies that supported me throughout my research. I thank Prof. Daniel Saab without whose collaboration and software tools my PhD would have been a lot more difficult.

My experience in Austin was enriched by the other members of the "AAA" gang (Sibi, Suriya, Vijay). I had a lot of fun interacting with them over the years. I hope to continue to do so.

Saving the best for the last, I would like to thank my family for their continued support and understanding over the years. My parents have been the pillars of my life, while my sister has been an inspiration for me with her own struggles for a PhD. I would also like to thank "Kutti" Bhavana for providing support and entertainment while I wrote the dissertation.

Automatic generation of instruction sequences for software-based self-test of processors and systems-on-a-chip

Publication No. _____

Sankaranarayanan Gurumurthy, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Jacob A. Abraham

At-speed functional tests are an important part of the manufacturing test flow of processors. With delay defects becoming more common due to the properties of the newer process technologies, at-speed functional tests have become indispensable.

Traditionally, functional tests needed expensive automatic testing equipment due to the memory and speed requirements associated. This cost issue was solved by native-mode testing which uses the intelligence of the processor to test itself. In the native-mode self-test (also known as software-based self-test) paradigm, instruction sequences are loaded into the cache to test the processor for defects. Generally, only random instructions are used in native-mode tests. As with any random sequence based testing, there are faults that are left undetected by random instructions. Manual effort is necessary to generate the tests that can detect those faults, requiring a detailed knowledge of

the instruction set architecture and the micro-architecture of the processor. We propose an automatic technique that alleviates the need for such manual effort.

Our technique has a hierarchical approach. We use traditional automatic test pattern generation (ATPG) algorithms for generating tests at the local level (module or combinational blocks). These tests are mapped to instructions at the global level using a verification engine. We also have feedback between these two levels for more efficient testing. We demonstrate the technique on a publicly available processor. We then enhance the technique to test an entire system-on-a-chip (SOC). A typical SOC has an embedded processor. We use this embedded processor to test the other blocks in the SOC. In general, most of these blocks are designed by the design reuse methodology. Therefore, these blocks may be available only as black-boxes. Our technique is well suited to test such blocks. We use existing test vectors for the core and present a technique that generates instruction sequences that when executed by the processor generates the given vectors at the boundaries of the blocks. We designed an SOC using an ARM core and a publicly available encryption core and experimented on it to demonstrate the effectiveness of our technique.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The process of transforming a design from an abstract implementation to a physical circuit on silicon contains various stages. At each stage an examination is done to affirm whether the design is still the same as the one in the previous stage. Design validation and manufacturing testing are two such examination steps. Design validation checks whether the register transfer level (RT-Level) design satisfies the properties and constraints specified by the architecture/design document. Manufacturing tests, as the name suggests are done after the production of silicon. They assume that the design is correct and check whether any defects were introduced by the manufacturing process on the silicon. In this research, we concentrate on the manufacturing tests.

Manufacturing tests involve loading an automated test equipment (ATE) with test vectors, connecting the device under test (DUT) to the ATE, applying the tests to the DUT and collecting the responses from the DUT through the ATE. This raises two questions.

1. How are the tests generated?

2. How can we know that the tests run on the DUT are sufficient to detect all the likely defects introduced during the manufacturing process?

The tests are generated using a logic representation of the circuit. Many algorithms have been developed [28], [24], [41] for this purpose starting with the D-algorithm [67]. Satisifiability solvers have also been used for test generation [49]. Since the goal of the tests is to detect as many defects as possible, test generation should be targeted towards likely defects. However, the defect information is not generally available at the logic level. Therefore, many fault models (like stuck-at, path delay) have been proposed that model the physical defects at the logic level. Depending on the complexity of the fault model, many faults may be introduced on the logic representation of the circuit. Each fault will define a corresponding faulty circuit at the logic level. The test generation programs will have to generate vectors that cause the good (fault-free) and faulty circuits to behave differently from each other. To know when the generated tests are enough, a metric called *fault coverage* was created. The fault coverage of a test is the ratio of the number faults detected by it to the total number of faults defined by the fault model on the circuit. The effectiveness of the tests is measured by their fault coverage.

One of the main differences between vector generation for design validation and test generation for manufacturing tests is based on the visibility of the internal signals. Design validation, as mentioned earlier, checks whether the RT-Level representation of the circuit conforms to the architectural specifications. Therefore, design validation test generation methodologies work with the RT-Level representation of the circuit. These tests are also applied (simulated) at the RT-Level, hence, during design validation all the signals

in the design are generally visible. For manufacturing testing, tests are generated using a logic representation of the circuit. However, they are applied on the silicon where only the design inputs/outputs are visible. Therefore, the test generation algorithms for manufacturing tests will have to guarantee both *controllability* and *observability*. *Controllability* ensures the excitation of the fault. For example, in case of a stuck-at zero (s-a-0) fault on a line, the test will have to produce a '1' on the line. *Observability* ensures the propagation of the faulty value to an observable point like a primary output. The test generation process has become increasingly difficult with the increasing complexity of designs. Moreover, the amount of testing needed to attain good coverage has also increased considerably. This, in turn, increases the amount of memory required on the ATE equipment. ATEs also have to be faster to due to the increase in clock frequencies. ATEs that can handle such at-speed tests cost millions. Design for test (DFT) [22] techniques like scan chains and built-in self-test (BIST) [34] were introduced to overcome these issues.

1.1 Scan and BIST techniques

A divide and conquer approach was suggested in [22] to make testing easier by introducing scan chains in the circuit. Scan chains involve modifying the memory elements (flip-flops or latches) in the circuit so that they can be directly loaded with values from an external environment. These memory elements are serially connected together. The testing now has two phases *shifting*, when the values are serially shifted into the memory elements, and

launch, when the values are launched from the memory elements into the circuit. The response of the circuit is captured by the scan elements. These values are then serially scanned (shifted) out. The test generation algorithm now handles only the combinational portion of the design. Hence, in general, both controllability and observability are improved. The values are shifted in and out at far lower speeds than the actual clock frequency. This allows the use of a slower ATE. Various test compression techniques were also proposed [38] that work in conjunction with the scan techniques to reduce test content and the test application time. Hence, the memory requirement on ATEs is also reduced, considerably lowering the cost.

The BIST [34] paradigm proposes introducing structures within the design for the purpose test application and response capture. BIST architectures were proposed for both the logic and memory. STUMPS [4] is a commonly used BIST architecture for logic. Memory BIST architectures are more popular and are generally based on *march* algorithms [30]. When logic BIST is used, the response of the circuit is captured in a linear feedback shift register (LFSR) or a multiple input shift register (MISR). These circuits capture the response and generate signatures that can be compared against the golden values.

1.2 Need for at-speed functional tests

With the advent of DFT techniques, the original methodology of testing without modifying the design came to be known as functional testing. Even

though using DFT techniques reduced the burden on the test generation tools and the ATEs, at-speed functional tests very never fully replaced. Even with the usage of multiple fault models there are certain defects that remain unmodeled and functional tests are good at detecting those unmodeled defects. Defects that affect the performance of a chip can only be detected by at-speed tests. Moreover, DFT techniques involve area overhead on the device. Any modification to a memory element makes it larger and slower. Hence, scan elements cannot be used on critical paths within the design. The vectors generated by BIST circuits do not have high coverage and a lot of vectors are needed, increasing the test time. Moreover, both scan and BIST vectors might lead the device into illegal states. State-of-the art designs are optimized for power and timing, but only for legal states. An illegal state might lead to high power consumption. There might be a lot of toggling when the test vectors are shifted in, this might also lead to an increased power consumption. A lot of research has been done towards optimizing power consumption during the DFT-based test mode [53], [91]. However, it is still an open issue. Moreover, with decreasing process geometries newer kind of defects are becoming more common [1]. Defects like resistive shorts and opens are not effectively tested by scan or BIST techniques [25]. The device operation also varies from chip to chip due to the parametric variations induced by the nanometer scale process technologies [11]. These variations manifest themselves as delay variations in the circuit. Both resistive defects and the process variations make delay defects (defects that cause change in the delay of the circuit) more common. It

has been shown that testing for delay defects using only scan/BIST based tests might cause yield loss due to overtesting [47]. All these factors make at-speed functional tests increasingly indispensable.

1.3 Native-mode self-tests

Traditional at-speed functional tests of processors involve running normal instruction sequences to test the processor. These instructions test the functionality of the processor [77]. For these tests, the processor is attached to an ATE which simulates the presence of the memory system and the other environments that a processor generally interacts with. The stimuli (functional tests) is applied by the ATE, which also gathers the response. These stimuli, in general, were made up of the architectural validation vectors and some legacy tests. However, as mentioned earlier ATEs that are capable of handling these tests at-speed are prohibitively costly. *Native-mode self-test* [72] provided a way to overcome the cost issue.

Native-mode self-test takes advantage of the processor's intelligence to test itself and is also called *software based self-test (SBST)* because it replaces most of the BIST hardware with software code. For example, an LFSR can be implemented in software. We will use the terms native-mode self-test and software-based self-test interchangeably through out this dissertation. The software programs used to test the processor can be directly loaded into the processor's cache from the ATE and run from there, requiring minimal design modifications. The loading of the cache can done at a lower speed than

the processor's clock frequency. Therefore, a lower speed ATE can be used, reducing the cost of the ATE. The block diagram showing the flow of native-mode self-tests is shown in Figure 1.1(b). However, the issue of selecting the kind of instructions that have to be used in the native-mode tests is a difficult problem. One way to handle it is to fill the cache with random instructions. However, there will be a lot of faults that are not detected if only random instructions are used. Manual effort has been required till now to detect those faults. It requires extensive and in-depth knowledge of the design/architecture to manually generate tests for those faults. Therefore, an automated technique to generate native-mode tests targeting specific faults has become a necessity. This is one of the primary focuses of the research presented in this dissertation.

1.4 Generation of instruction sequences targeting specific faults

We need an automated technique to avoid the intense manual effort involved in generating instruction sequences targeting specific faults. We cannot use traditional ATPG algorithms "as is" due to the way they operate. In general, they start from a specific fault and go forward and backward trying to propagate and justify it. Some of the ATPG tools have the ability to take in constraints. However, they tend to generate a vector first and then check whether the vector meets the constraints. It is very important that, in case of native-mode self-tests, the test vectors generated conform to the constraints

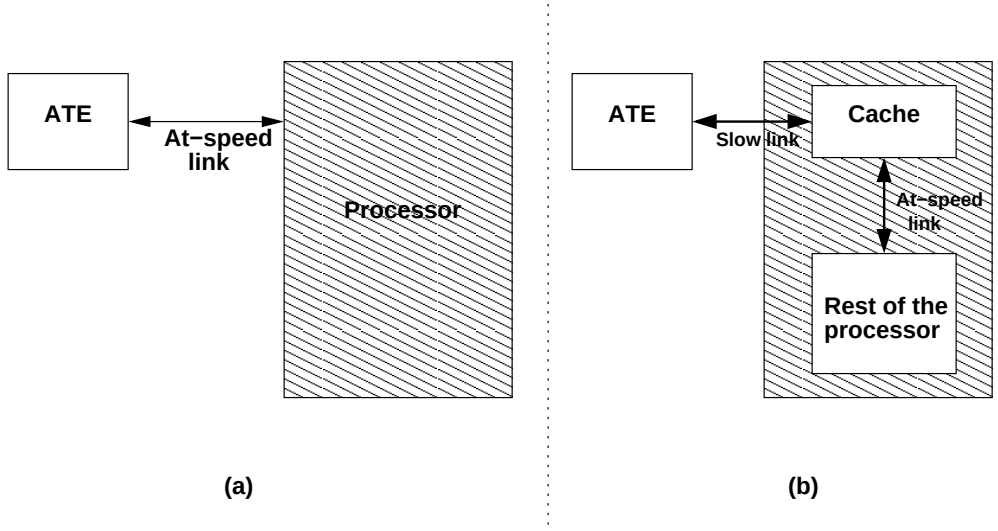


Figure 1.1: Illustration of functional testing techniques. a) Traditional method requiring an at-speed ATE b) Native-mode testing methodology reducing the speed requirement on the tester equipment

specified by the instruction set architecture (ISA). The constraints specified by the ISA can be very restrictive. For example, an ISA with 8-bit opcodes might specify only 100 valid opcodes making it likely that an 8-bit value generated without the prior knowledge of the ISA will not be a legal opcode. Therefore, using traditional ATPG algorithms generally gives very low coverage when trying to meet the constraints set forth by the architecture. However, the ATPG tools are very efficient in generating tests for gate-level faults. Therefore, we propose a hierarchical approach that includes an ATPG tool at the lower level to make use of its advantages, while using a different tool at the higher level.

The hierarchical approach as shown in Figure 1.2 uses traditional ATPG algorithms at a local level and a verification engine at a global level. The local

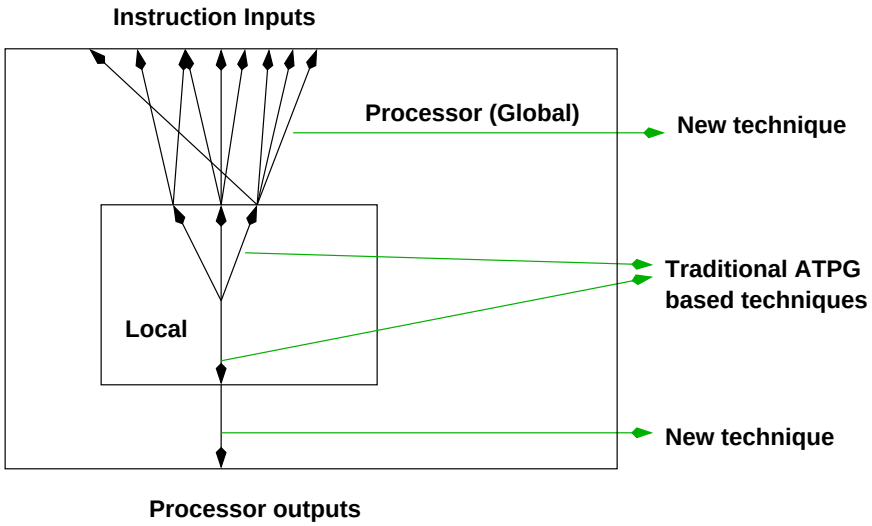


Figure 1.2: Hierarchical operation of the proposed technique

level may be a module of the design or a combinational block. The global level contains the entire processor core. The verification engine we use is a bounded model checker. A bounded model checker takes in as inputs a property, a design and a bound. It unrolls the design to the bound and checks whether the property can be disproved within that bound. The advantage that this verification engine provides is the fact that it can take in the constraints from ISA and use it as part of its search process. Therefore, unlike an ATPG algorithm that generates a vector and then checks whether it is within the ISA, the verification engine will start with the constraints we specify (like the ISA) and then search for the vector. Verification engines are generally meant for design validation and are not suited to take in the fault information and generate tests for those faults, making them ineffective at the local level. In effect, by using different engines at different levels, we try to get the best of

both worlds.

In the following chapters we will go into more detail while explaining the technique and the experiments that we did. We first apply our technique to stuck-at faults. We use a commercial ATPG tool at the module level. We then map the sequence generated by the ATPG tool to instructions using the bounded model checker. The ATPG tool generates test sequence at the inputs of the module and propagates the faulty value to a module output. The mapping technique, using the bounded model checker, both maps the test sequence at the module inputs and propagates the faulty value from the module output to an observable output. The bounded model checker is given the RT-Level design to work with. For the propagation of faulty value, we initially propose an iterative refinement technique. There is a manual component involved in the iterative refinement. We do away with this manual component later by introducing an entity called *observation abstraction*.

1.5 Targeting delay defects

After targeting stuck-at faults, we apply our technique to test for delay faults. We use a similar hierarchical approach. However, in case of delay faults, each fault actually represents a path. A long path might extend over more than one module. Therefore, local level test generation cannot be stopped at the module boundaries. Hence, we define the combinational portion of the design as the local level during the test generation. Generating tests for a path and then mapping them to instruction sequences is not very efficient. Therefore,

we map the paths directly, making it necessary that the verification engine work with the gate level design. Moreover, attempting to map all the paths is not very advisable. Therefore, we have a local level test generator to act as a filter. We use a PODEM-based test generation tool on a delay annotated circuit to identify paths that have a test at a combinational level. The tool also identifies those paths that are critical as specified by a fault model. We use the global mapping technique to generate instructions to test those paths. A common non-functional segment might cause many paths to be non-functional. Many of the critical paths share common segments. Therefore, we develop a feedback mechanism which weeds out the paths that contain a sub-path that has been found to be non-functional. This helps us in reducing the amount of effort spent on mapping paths.

In case of delay paths, the effect of the path has already been propagated to a flop. It makes it more possible that any instruction sequence that guarantees the excitation of the fault will also incidentally ensure propagation of the fault to an observable output. Therefore, we propose a technique which generates excitation only tests and checks for propagation through simulation. We use a technique based on netlist modification which checks whether a test propagates a given path to an observable point. The number of possible faults in a circuit is higher than the number of stuck-at faults when a good delay fault model is used. Therefore, the number of tests required to achieve good coverage in the case of delay faults may be significantly higher. Hence, we enhance the simulation technique to identify all the critical paths that are

detected by each test. This helps us in reducing the test data.

1.6 Testing a system-on-a-chip

As the ITRS document of 2005 notes [1], there is an increasing trend towards integration of designs making systems-on-a-chip (SOCs) more common. Therefore, testing of SOCs has become a pressing issue. In general, techniques like boundary scan [87] are used while testing SOCs. These techniques have similar drawbacks as the DFT techniques mentioned earlier. Therefore, it is imperative that these tests be complemented with functional tests. Most SOCs contain an embedded processor. We develop a solution that uses this embedded processor to test other cores in the SOC.

In SOC designs, the concept of *design reuse* is very prominent. Design reuse involves using blocks of design that already exist instead of designing a whole new chip. Blocks of logic called intellectual property (IP) cores are provided by many vendors that are used by the designers while developing an SOC. These cores are of three kinds.

1. White-box – Internal implementation of the core is visible. Changes can also be made to the core.

2. Grey-box – Internal implementation of the core is visible. Changes cannot be made to the core.

3. Black-box – Internal implementation of the core is not visible. Changes cannot be made to the core.

Many of the IP cores are available as black-box only, which means that the SOC designer will not know the internal implementation of the SOC. Any SOC testing technique that caters to black-box cores should work without the knowledge of the logical implementation of the cores. Vendors providing IP cores may also provide test stimuli that can be used to verify/test the core. We make use of these vectors. We map those stimuli into instructions. These instructions when loaded into the embedded processor, will produce the stimuli at the inputs of the core. The advantage of this method is that it can be used to test grey-box and white-box IP cores too. In those cases, we can also have a hierarchical method, where we generate test vectors for the IP cores by using their logical implementation and then map them using our technique.

1.7 Outline

In the following chapters we explain our techniques in detail. Before going into detail about our technique we survey the relevant work in the testing paradigms that are related to the research presented here. The testing paradigms that are relevant to our work include hierarchical testing, functional testing for delay and stuck-at faults, SOC testing and RT-Level testing. We provide a sample of the work available in literature in these fields in Chapter 2.

We explain the automation of the controllability part of the mapping technique in Chapter 3. The controllability part of the mapping technique pertains to the excitation of the fault with an instruction sequence. We also elaborate on the iterative refinement technique for propagation in this chapter.

We use the stuck-at fault model as our target fault model and use the opencores RISC (OR1200) processor as our design for experimentation in this chapter. In Chapter 4, we present our technique of using observation abstractions for automated propagation of faulty values. We again use stuck-at fault model and the OR1200 processor. We chose OR1200 because it is publicly available from opencores website [89] and is a fairly complex design.

We will explain our technique for testing for delay defects in Chapter 5. In this chapter, we will use a gate level model of the OR1200 processor synthesized using standard cells which have been characterized for delay. The fault model that we use for delay defects is a superset of the stuck-at fault model. Therefore, the coverage results presented in this chapter are valid for stuck-at faults also. We then move on to generating efficient tests, *i.e.*, we generate instruction sequences that have better probability of detecting multiple delay faults. Our technique for doing so is presented in Chapter 6.

We design and develop a SOC using an ARM processor and an advanced encryption standard (AES) core. We apply our technique for SOC test generation on this model. We show the results of this and explain the SOC testing technique in detail, in Chapter 7. We provide illustrative examples while explaining all of these techniques. We present the conclusions that we drew from the research in Chapter 8. We also discuss strengths and drawbacks of our techniques in that chapter. We also lay some groundwork for future research in related areas in Chapter 8.

Chapter 2

Testing: Various approaches and issues

2.1 Functional tests

Many approaches have been presented to tackle the problem of functional testing. Earlier methods did not target structural gate level faults. The fault models used were functional. In the seminal work in this kind of testing, Thatte et al. [77] proposed having a graphical representation of the functional description of the processor and generating tests (instructions) based on this graph. A functional fault model depicts various functionalities of the processor like register decoding and instruction execution. This methodology was enhanced later to include complex instruction execution and cache access in [12] and [29]. A control fault model was proposed at the instruction level in [73]. This method considers the read/write instructions which setup the test separately and has a checking experiment for those instructions. The main drawback of using a functional fault model is based on the fact that it might not have a good correlation with the actual defects caused by the fabrication process and test generated with good coverage of such a model might still miss some defective chips.

A new methodology for functional testing was proposed by Shen et

al. in [72]. They proposed loading random instruction sequences into the cache and testing the processor using those instructions. This methodology was applied in an industrial setting in FRITS [65]. FRITS detected many defective chips that passed traditional tests. This result has propelled the recent research in this area. Many techniques were proposed with the gate level fault model as the goal, mostly targeting the stuck-at fault model. Kranitis et al. [42] presented a method to generate deterministic programs for self-testing of arithmetic modules. This method, however, requires knowledge of the functionality of each block within the RT-Level implementation. Corno et al. [19] used evolutionary techniques to direct the search process. Chen et al. [17] dealt with mapping of module level sequences to instructions. Their technique extracts module signals and maps them to *instruction templates* using pre-defined mapping functions. However, their technique depends on the quality of the instruction templates and the mapping functions. All these techniques target stuck-at faults, however, they do not, in general, have the capability of guaranteeing tests for specific faults.

2.2 Hierarchical testing

The techniques that we propose for testing processors and SOCs have two levels and are hierarchical. A hierarchical approach has been popular for sequential test generation because it makes it possible to target complex designs due to the divide and conquer nature of such an approach. Moreover, most of the state of the art chips are designed by multiple designers. Therefore,

the designs are split into multiple blocks with various blocks communicating with each other. The modularity of such designs makes a hierarchical approach highly attractive. The test generation for processors also lends itself for a hierarchical approach because it has to work under two different kind of constraints. At the global level it has to deal with the constraints imposed by the ISA and the external environment, and at the local level it has to satisfy the constraints caused by the structure of the circuit and the faults targeted.

Roy et al. [68] proposed using the data flow descriptions to generate tests for sequential VLSI circuits hierarchically. They use a recursive approach for propagation and justification. An automatic knowledge extraction tool (ATKET) was presented in [84]. This tool extracts the information from various modules into suitable data structures and generates tests using a custom ATPG tool. The ATPG tool will always maintain information about various modules in the design. Murray and Hayes [61] presented a two level approach where they generate module level test sequences and then generate tests for them at the full chip level. They term these module level test sequences as *pre-computed sequences* and use them as primitives. A hierarchical approach to test processors was proposed by Tupuri et al. in [79]. Their approach extracts the constraints imposed by the circuit on a module of the design and proposes implementing those constraints as virtual constraint circuits around the module. It proceeds to use an ATPG engine to generate tests for faults in the module using the module and its corresponding virtual constraint circuit. This technique was extended to SOCs in [81]. A program slicing based ap-

proach was suggested by Vedula et al. [82]. Their approach uses slicing at the RT-Level to remove all the statements which are not related to the module under test. This reduced code is then supposed to be synthesized and given to an ATPG tool.

2.3 RT-Level test generation

The technique that we propose for stuck-at faults specifies the properties for the verification engine at the RT-Level. We can therefore use an engine that works at the RT-Level. Generating tests at the RT-Level has many advantages. The code at the RT-Level is more intuitive and simple. Since the complexity of the design is lower at the RT-Level, test generation might be easier. In most hierarchical approaches the global level test generation does not need the circuit level information. Therefore, the global test generation can be generally done at the RT-Level. If a gate level fault model is not used, RT-Level test generation can be used even without a hierarchical approach. However, those fault models would have to be correlated with the gate level fault models.

An example of using RT-Level test generation in a hierarchical approach is [61]. In that approach, Murray et al. generate tests for pre-computed test sequences at RT-Level. However, their technique requires the design to be acyclic. Bhatia et al. [7] proposed a RT-Level test generation solution that requires the control and datapath to be separate. Lingappan et al. [52] represent the RT-Level design as algebraic decision diagrams (ADD) [2] and use

abstractions to represent the components of the ADD. Fallah et al. [23] proposed a method which generates tests for observability enhanced code coverage of hardware description language (HDL) descriptions. They use a hybrid method which employs both linear programming and Boolean satisfiability solvers. Xin et al. [88] generate tests for behavioral VHDL descriptions. Their method works for a highly restrictive form of VHDL code.

2.4 Targeting delay faults

With the decreasing geometries of the process technologies, delay defects are becoming more common [25], [11]. Thus, testing for delay defects is a must in future chips. In general, at-speed functional tests are known to be very effective in detecting delay defects [25]. Moreover, using techniques like enhanced scan [21] to test for delay defects might lead to yield loss due to overtesting [66]. It has been shown that a large portion of the paths in the design ($\approx 80\%$) are functionally untestable [47]. All these factors have caused an increased interest in generating functional tests for detecting delay defects.

Lai et al. proposed a technique in [48] to target path delay faults. However, their methodology targets only non-pipelined processors. Singh et al. [74] target path delay faults in pipelined processors. However, their technique involves manually constructing a graph model for the processor. Moreover, they do not tackle the problem of propagating the effects of faults from internal flops (non-register file flops) to observable outputs. Lin et al. proposed a technique for path delay fault testing of processors in [51]. They developed a

methodology called pseudo-functional BIST. This technique does not generate functional sequences, instead it uses scan based testing and increases the probability of scan vectors to be functionally feasible.

We will deal with delay fault simulation in order to reduce the test content. In general, commercial ATPG tools do not provide fault simulation of path based delay faults for sequential circuits. [16], [18], [62] all provide fast memory efficient ways for path delay fault simulation. However, they are oriented towards scan based tests. A methodology based on netlist modification was explained in [6]. This technique involved sabotaging the netlist so that if path under test is excited, a faulty value is latched into the memory element at the end of the path. This technique can be used for fault simulation when functional tests are involved.

2.5 Testing SOCs

SOCs are becoming more common making testing them an important issue to be addressed. In general, SOCs contain an embedded processor core. Most of the functional test approaches surveyed in Section 2.1 can be used for testing the embedded processor core. Many approaches have been proposed for testing the other cores. SOCs are generally designed using the design reuse paradigm. The non-processor cores in the SOC are mostly bought from intellectual property (IP) vendors. Many of these cores (known as IP cores) are available only as black-boxes. In general, test sequences already exist for these cores either from the vendor providing the core or from a previous use

of the core. Therefore, most of the research on testing the IP cores is focused on providing test access mechanisms (TAMs) to deliver the test sequences.

The TAMs can be based on direct access where the cores can be accessed through pins of the SOC. Direct access can be boundary scan based [78], [87]. These kinds of TAMs provide a serial access. Therefore, the test application time will be high due to the time taken for serial scan shifting. This can be reduced by providing a parallel access to the IP cores [37]. However, such a scheme would entail allocation of many SOC pins for test access and also area overhead for the routing of access mechanisms.

Test access can also be done through the embedded processor cores. The access from the embedded core can be non-functional, *i.e.*, this kind of access will not occur during the normal operation of the SOC. Papachristou et al. [64] proposed direct access to the cores from the processor and also the responses to be collected in a signature verifier. Krstic et al. [45] proposed having test wrappers around the IP cores and accessing the wrappers through the processor. Hwang et al. [36] provided a similar technique based on reuse of the addressable system bus. The test access through the processor can be functional, *i.e.*, the processor will access the cores through mechanisms that it uses during the normal SOC operation. The advantage of this kind of access is that there is very little area/power overheads involved. Jayaraman et al. [40] proposed a way to test USART with such an access. Tehranipour et al. [76] used functional access to test the peripheral cores. Their approach needs information on processor and peripherals so that instructions can be sorted

and grouped according to their likelihood of accessing various parts internal to cores. A methodology using the high level coverage metric was proposed to test the peripherals in [9], [10].

Chapter 3

A two level approach for generating functional tests

When random instruction sequences are used in functional tests, there is a class of faults, termed as *hard-to-detect* faults, that is not detected. In order to target such hard-to-detect faults, it is beneficial to have knowledge about the instructions that can excite those faults. Although generic sequential ATPG tools ([5],[63],[70]) are highly optimized, they are not effective for large designs like processors. These ATPG tools can, however, be used effectively to generate test sequences for faults at the module level. These module level test sequences can then be mapped to primary inputs/outputs of the processor. The input space of the processor is decided by its instruction set. Therefore, the module level test sequences can be mapped to instruction sequences.

In this chapter, we present a novel hierarchical approach to generate instruction sequences targeting specific faults in a processor. We use an ATPG tool to generate tests at the module level and then map the module level test sequences globally to instructions using a verification engine. Our global mapping works at the RT-Level. The verification engine we use is based on bounded model checking [8]. Our technique is generic and can be applied to

any off-the-shelf processor. It is intended for use by the designer and requires minimal expertise.

Bounded model checking is used for both controllability and observability analysis of module level test sequences. Controllability of a given test sequence is dealt with in the following way. We generate a Linear Temporal Logic (LTL) [54] safety property for controllability from a given module level test sequence. Cadence SMV's bounded model checker (SMV-BMC) [46] is then used to verify the generated property. We also constrain the SMV-BMC input space by providing it the instruction set of the processor. The property is written such that if the module level test sequence can be generated through the instruction set, then SMV-BMC provides a counterexample for the property. We term the test sequences that can be generated through the instruction set as *functionally feasible*. The counterexample generated by SMV-BMC will contain a possible instruction sequence that can produce the given module level test sequence. Observability of the module level test module level test sequences is handled in a similar manner, by applying SMV-BMC iteratively.

The main contributions of this chapter are as follows.

1. We introduce a technique for mapping module level test sequences to instructions, using bounded model checking.

2. We provide an iterative method to check the observability.

3.1 Background

3.1.1 Bounded model checkers

When a model and a property are given to a model checker, it will give a yes/no answer that tells us if the property holds on the given model or not. If the property does not hold, an error trace, or a counterexample is produced. Bounded model checkers prove the correctness of properties of a model within a given bound. We use SMV-BMC, which accepts properties written in LTL.

3.2 Instruction mapping of module level test sequences

We apply bounded model checking to instruction mapping of module level test sequences for every module of the processor's design. For every module of the processor, we translate a module level test sequence into an LTL property. The property ties the signals to the corresponding values in the module level test sequence. *We negate this property and pass the resulting safety property through SMV-BMC.* If a counterexample is produced, it implies that the signal values given in the module level test sequence *can* be generated by an instruction sequence of the processor.

The connection between the module inputs and the processor inputs (instruction sequence) is made due to the hierarchical operation of SMV-BMC. Figure 3.1 shows this operation pictorially. In Figure 3.1, the solid arrows show the input/output dependencies between modules and the dashed arrows give the hierarchical flow of a counterexample. M3 is the module under test (MUT). The module level test sequence at the inputs of M3 is converted into a LTL

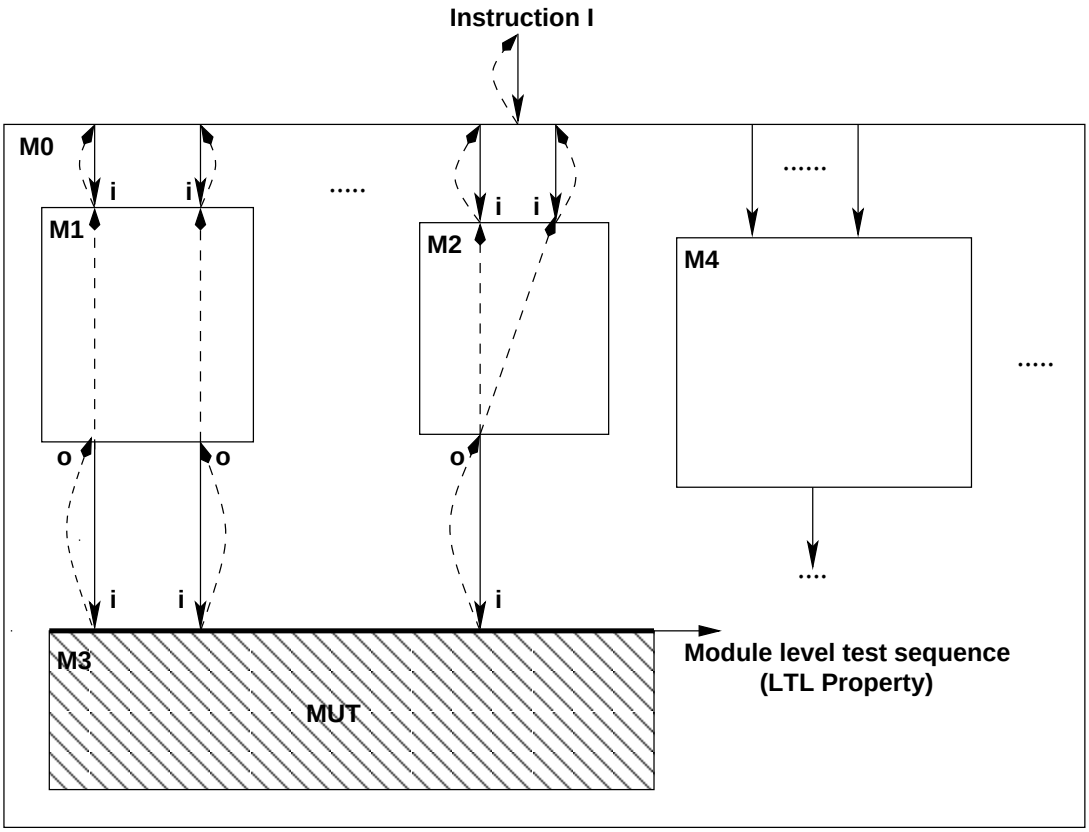


Figure 3.1: A hierarchical structure given to SMV-BMC. The dashed arrows give hierarchical flow of counterexample and the solid arrows show the input/output dependencies between modules.

property and passed through SMV-BMC. The counterexample, if generated by SMV-BMC, contains the values of outputs ('o' in the figure) of M1 and M2. The counterexample also provides values of all the intermediate signals of M1 and M2 (as shown by dotted arrows) and the inputs ('i' in figure) of M1 and M2. Therefore, the counterexample provides signal values until the instruction 'I', which is the input of the main module M0. We constrain the input space of SMV-BMC by providing it the instruction set of the processor

under test. SMV-BMC is also used for extracting observability constraints and checking the functional feasibility of module level test sequences. A bound is computed and given to SMV-BMC. The computation of the bound is natural in the case of in-order pipelined processors with the external stalls disabled. In such a processor an instruction which enters the pipeline exits after N cycles, where N is the pipeline depth of the processor. Therefore, the effect of the instruction is felt for $N + 1$ cycles if there is pipeline forwarding and for N cycles if there is no forwarding. Hence, the bound should be more than the pipeline depth of the processor. If there are multicycle instructions, then the number of extra cycles taken by the longest instruction should also be added to get a good bound.

Figure 3.2 gives the flowchart for the technique. The shaded boxes in the flowchart are implemented using SMV-BMC. We start with a module level test sequence and first analyze its controllability. In general, controllability deals with the possibility of generating a desired value for an internal signal. We check if the given module level test sequence can be generated through the instructions in the instruction set of the processor. We need to define controllability property C for this purpose.

The controllability property C is defined such that, if it fails, the counterexample generated by SMV-BMC will contain the desired instruction sequence at the inputs of the processor.

For example, the test sequence given in Table 3.1 is first translated to the LTL property:

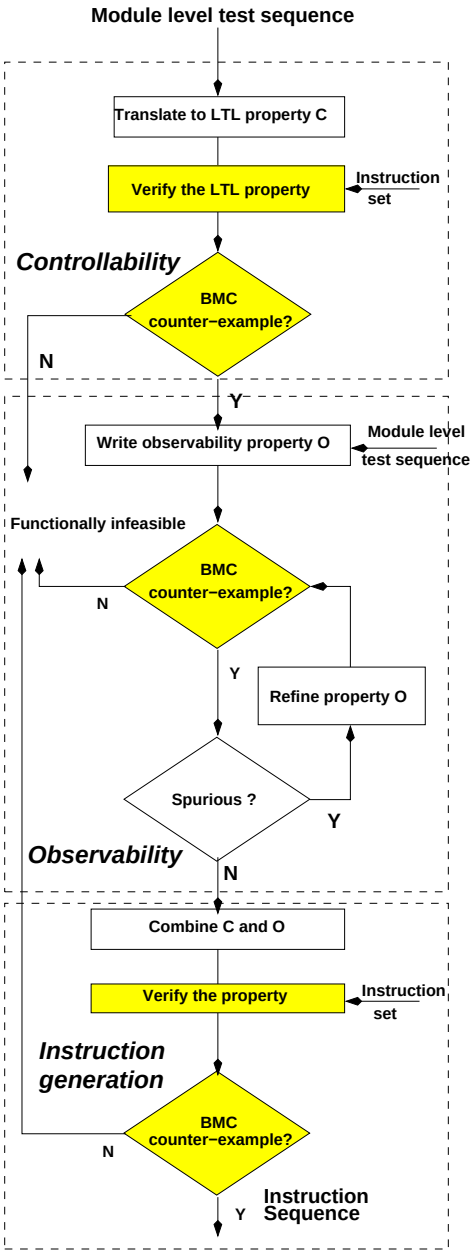


Figure 3.2: Flowchart for the instruction mapping of module level test sequences. The shaded boxes are implemented using SMV-BMC

Table 3.1: An example test sequence

time	a	b	c
0	1	0	x
1	1	1	x
2	x	x	1

```
always
  if ((a == 1) && (b == 0)) begin
     wait(1);
     if ((a == 1) && (b == 1)) begin
         wait(1);
        if (c == 1) begin
          //property holds
           assert P: ‘TRUE;
        end
     end
  end
```

This property is then negated and built into an assertion. This assertion is the desired controllability property C. For the example test sequence, C is:

```
always
    if ((a == 1) && (b == 0)) begin
      wait(1);
     if ((a == 1) && (b == 1) begin
       wait(1);
        if (c == 1) begin
          assert property: ‘FALSE;
        end
     end
    end
```

The property states that a, b and c can never have the values given in the test sequence. The values in times 0, 1 and 2 are modeled using the next state operator in LTL, which is represented by the `wait` statement in Verilog.

We pass the controllability property C to SMV-BMC along with the processor's RT-Level source code. SMV-BMC is also given the instruction set

of the processor under test. The bound for SMV-BMC is calculated using the method described earlier.

SMV-BMC checks for the existence of a counterexample for the stated property within the given bound. If no such counterexample exists, then the module level test sequence is functionally infeasible (within the given bound), since it is not controllable. The technique proceeds only if there is a counterexample.

The module level test sequence also identifies the module outputs that need to be propagated to observable points to make it observable. Therefore, in the observability stage of the technique, we extract constraints needed to propagate these module outputs to an observable point of the processor. The observable points of a processor include primary outputs, memory, register file and all other user accessible points of the processor.

Since we use SMV-BMC for observability, we define an LTL property for observability. The observability property O is defined such that if it fails, the counterexample generated by SMV-BMC will contain the constraints necessary for propagation of the module outputs. Therefore, observability property O states that a change in a module output signal should not cause a change in any of the observable points of the processor. This property is of the form:

```
always
 begin
    temp_mo = mo;
    temp_Po1 = Po1;
    ...
    temp_PoN = PoN;
```

```
   wait(1);
   if (mo ^ temp_mo)
      assert O: !(Po1 ^ temp_Po1)
                & ...
                & !(PoN ^ temp_PoN);
end
```

where `mo` is the module output that needs to be propagated and `Po1,Po2,...PoN` are the observable points.

The property states that a change in the value of `mo` in the next state does not imply that eventually one of the observable points will change their value. A change is modeled by using an xor operator between the signal value in the current state and the next state.

Property O is passed to SMV-BMC. If no counterexample is generated, then the test sequence is functionally infeasible since it is not observable. However, existence of a counterexample does not necessarily prove that module output is observable. For example, consider the Verilog model given below:

```
always @(a or b or c or d or e)
 begin
 ...
  if (d == 1)
   c = a;
  else c = b;
   f = c | e;
 end
```

Let `a` be the module output which needs to be propagated and `f` be the only observable point. The initial observability property O would then be:

```
 always
  begin
    temp_a = a;
    temp_f = f;
    wait(1);
    if (a ^ temp_a)
    assert O: !(f ^ temp_f);
  end
```

It is possible then that SMV-BMC generates the counterexample:

```
state 1:        state 2:
a = 0           a = 1
b = 1           b = 0
d = 0           d = 0
c = 1           c = 0
f = 1           e = 0
                f = 0
```

The counterexample shows a change in the value of `a` and a change in the value of `f`, which disproves the property O. However, `a` has not actually been propagated. We term such a counterexample as *spurious*. A counterexample is spurious if it disproves the observability property, although it does not guarantee propagation of a module level test sequence. Therefore, the property O has to be refined, to generate the correct counterexample, and thereby, the correct constraints. For the given example, O is refined by adding the constraint that the value of `d` should be set. Hence, the refined property O is:

```
always
 begin
   temp_a = a;
   temp_f = f;
   temp_d = d;
   wait(1);
   if ((a ^ temp_a) & d & temp_d)
     assert O: !(f ^ temp_f);
 end
```

The counterexample generated for this property gives all the necessary constraints for observability. In general, the property O is iteratively refined by adding constraints based on spurious counterexamples, until a counterexample that actually models the propagation is found. The bound can also be iteratively changed after starting with pipeline depth as the initial bound. Note that the instruction set is not given as a constraint to SMV-BMC while checking for observability. There is no need to constrain the inputs to the

instruction set, Adding instruction set information to the property places the unnecessary constraint of finding an instruction sequence which will cause a change in the module output. This change in module outputs is caused only when there is a fault, and does not depend on instructions.

The extracted observability property O, is combined with the controllability property C using simple conjunctions, and passed to SMV-BMC. At this stage, SMV-BMC is also given the instruction set as a constraint. If SMV-BMC produces a counterexample then the instruction sequence is extracted from the counterexample. This is the required instruction sequence generated by our technique.

It is possible that SMV-BMC does not find a counterexample. This might be because of one of two possibilities. The first possibility is that the controllability and observability properties contradict each other, *i.e.*, the signal values obtained in the counterexamples for C and O have conflicting values. In that case, SMV-BMC eliminates the module level test sequence. The second possibility is that the observability constraints do not coincide with the possible instruction sequences. This is possible, because we extract observability constraints from SMV-BMC without providing it the instruction set as a constraint. If no counterexample is generated, the module level test sequence is declared as functionally infeasible.

3.3 Mapping module level test sequences to instructions for OR1200

3.3.1 OR1200

The OR1200 is a publicly available processor design. The source code in Verilog RTL of OR1200 is available from [89]. The specification manual can also be found at [89].

The OR1200 is a 32-bit scalar RISC processor, with a Harvard micro-architecture, 5 stage integer pipeline, virtual memory support (MMU) and basic digital signal processing (DSP) capabilities. The central processing unit (CPU) of the OR1200 has an instruction unit that implements the basic instruction pipeline. There are 32 general purpose registers (GPRs) of 32-bits each in OR1200. The load store unit handles all the transfer between GPRs and the internal bus of CPU. There is also an exception handling unit which implements a uniform procedure for all exceptions. The integer execution unit of OR1200 executes most integer instructions in one cycle. The basic block diagram for OR1200 is given in Figure 3.3.

3.3.2 Generating module level test sequences for OR1200

We have to generate module level test sequences for OR1200. In order to select faults in a module to target for the process, we wrote a random instruction sequence which has 36750 instructions. The OR1200 was fault simulated with this instruction sequence, and the fault coverage saturated around 68% for stuck-at faults. We split the undetected faults into separate

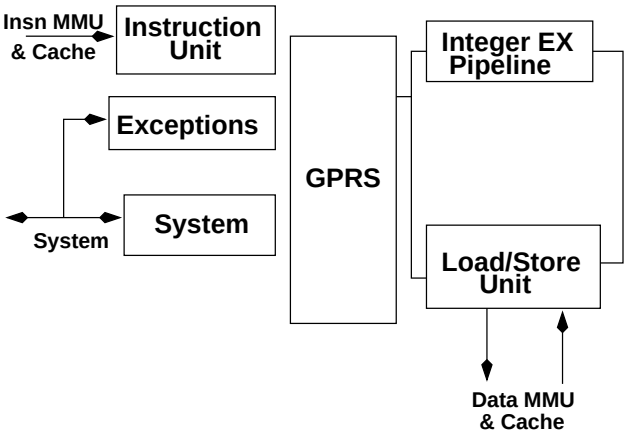


Figure 3.3: OR1200 CPU's block diagram

lists depending on the module to which they belong. A commercially available ATPG tool was used to generate module level test sequences for each of these faults. These module level test sequences were then fault simulated to add information about module outputs that would change their values, in case of a fault.

3.3.3 Generating LTL properties from module level test sequences

OR1200 has various external inputs that can cause a stall in the pipeline. This helps the processor communicate correctly with the environment. However, we need to ensure that the counterexamples generated by SMV-BMC do not stall the processor due to these external inputs. Similarly, we also need to ensure that the counterexamples do not reset the processor. We state the above constraints as part of our controllability property. The instruction set information is added to the property, to constrain its input space.

Setting those constraints for the properties is an one time operation. These constraints remain the same for all module level test sequences. Each module level test sequence is translated to its corresponding LTL property, and the constraints are added to it. For example, for the test sequence (for the module operandmuxes) given in Table 3.2, the controllability property is given in Figure 3.4.

Table 3.2: A module level test sequence for OR1200

time	id_freeze	ex_freeze	sel_b[1]	sel_b[0]
0	0	0	X	X
1	X	0	1	0

We have defined `IF_INSN_LEGAL` such that it is set if the instruction fetched in the current cycle is legal (part of the instruction set of the OR1200). As shown in the property `PR`, `illegal_insn` is set if an illegal instruction (not part of the instruction set) is fetched.

The remaining variables in the above property are the inputs of a module. The assertion `PR` fails if the module level test sequence is generated at the inputs of the module with legal instructions. Note that the ISA and the environmental constraints remain the same. The part of the property pertaining to module inputs is different in every case.

3.3.4 Observability

The initial observability property for any module level test sequence is similar to the one generated for the example given in Section 3.2. In some cases,

```
always
  begin
     // rst and stall deactivation
     icpu_err_i = 0; //for deactivating fetch stalls
     du_stall = 0;    //for du_stall
     dcpu_rty_i = 0;  //for lsu_stall
     rst = 0;

...

     // Test sequence and instruction set
     if (!(`IF_INSN_LEGAL))
            illegal_insn = 1'b1;
     if ( (`IF_INSN_LEGAL) &&
        (or1200_operandmuxes.id_freeze == 0)
        && (or1200_operandmuxes.ex_freeze == 0))
        begin
          wait(1);
          if ( (`IF_INSN_LEGAL) &&
           (or1200_operandmuxes.ex_freeze == 0)
           (or1200_operandmuxes.sel_b[1] == 1) &&
           (or1200_operandmuxes.sel_b[0] == 0))
           begin
             wait(1)
             assert PR: (illegal_insn == 1'b1);
        end
     end
```

Figure 3.4: OR1200 controllability property example

```
always
 begin
  if (or1200_alu.flagforw == 1'b0)
  begin
    wait(1);
    if (or1200_alu.flagforw == 1'b1)
    begin
      wait(1);    //wait till previous flagforw
                  //bit reaches register file
      temp = rf_dataw;
      wait(1);    //wait till later flagforw bit
                  //reaches register file
      if (rf_dataw != temp)
        assert : 'FALSE;
    end
  end
 end
```

Figure 3.5: Property for observability

the initial observability property does not ensure propagation of outputs to observable points. For example, for one of the module level test sequences, the `flagforw` output of the `or1200_alu` needs to be propagated to an observable point.

The initial property is shown in Figure 3.5.

`rf_dataw` contains the data written to register file and is an observable point, since we make the register file observable. However, this property is not enough to obtain the observability constraints. Consider the Verilog code given in Figure 3.6. `muxin_c` is the signal vector tied to `flagforw`. `muxout` is directly tied to observable outputs.

The initial counterexample has the most significant bits of `rfwb_op` set to 00. In this case `flagforw` is blocked from propagating further. Therefore,

```
always @(muxin_a or muxin_b or muxin_c or
                        muxin_d or rfwb_op)
case(rfwb_op[2:1])
 2'b00: muxout = muxin_a;
 2'b01: muxout = muxin_b;
 2'b10: muxout = muxin_c;
 2'b11:  muxout = muxin_d + 4'h8;
endcase
```

Figure 3.6: Verilog code snapshot

the property has to be refined to add the constraint that the most significant bits of `rfwb_op` need to be tied to the value 10. A counterexample that is not spurious is obtained after adding this constraint.

Therefore, the property for observability is iteratively refined to achieve the desired results. However, once the constraints are extracted for propagation of all the outputs of a module, all the module level test sequences for that module can be propagated using the same constraints.

3.3.5 Instruction generation

We have automated the generation of the controllability property as well as checking its functional feasibility. All the module level test sequences generated as described in Subsection 3.3.2 are checked for controllability. As shown in Section 3.4 many of those module level test sequences are functionally infeasible. Once a module level test sequence passes the controllability test, an observability property is defined for it and signal values for output propagation are extracted. The refined observability property is combined with the controllability property to get the overall property of instruction mapping.

The counterexample generated by SMV-BMC for this property contains an instruction sequence that would generate the module level test sequence. If no counterexample is found then the module level test sequence is declared as infeasible.

3.4 Experimental results

As mentioned in Section 3.3, experiments were performed on OR1200 processor [89]. A long pseudo-random instruction sequence containing 36750 instructions was generated for the OR1200 CPU core. These 36750 instructions were generated by randomly varying the data operands for the instructions of OR1200. The OR1200 CPU core was fault graded for all possible stuck-at faults for the random instruction sequence using a commercially available tool [20]. The fault coverage saturated around 68%. The list of faults that were left undetected formed the base list. We applied our technique on this base list. We thereby simulated hard-to detect faults which are the principal target of the technique.

We split the base list based on modules. We generated the module level test sequences by the method described in Section: 3.3.2. We checked functional feasibility for all the module level test sequences that were generated. Table 3.3 shows the results for the functional feasibility check of all the module level test sequences.

The first column in Table 3.3 gives the name of each module. The number of module level test sequences for each module (which is same as

Table 3.3: Result for feasibility check on module level test sequences

Module	No. of module level test sequences	Uncontrollable Test sequences
alu	2812	14
genpc	2964	2345
sprs	4626	3243
wbmux	1053	33
lsu	1401	87
freeze	85	23
ctrl	1620	437
except	5965	269
operandmuxes	421	35
if	1688	279

the number of faults in each module that were undetected by the random instruction sequence) is given in the second column. The number of test sequences that were functionally infeasible is given in third column.

Out of the module level test sequences designated as functionally feasible, we picked some sequences at random and we generated an instruction sequence for each of them. We kept track of observability constraints that were extracted for each module level test sequence and re-used them if the same module output had to be made observable for some other module level test sequence.

SMV-BMC was allowed to start from any state allowing registers to contain any value at the beginning of the instruction sequence generated by SMV-BMC. Therefore, we added an initialization sequence which loaded those values into registers. OR1200 has a load-store architecture. Therefore, it is

Table 3.4: Instruction mapping for each module

Module	No. of module level test test sequences	No. of Instructions
alu	9	18
ctrl	5	23
operandmuxes	5	19

easier to initialize the registers to desired values. Table 3.4 shows the results of our instruction mapping for the example modules. The results have been shown on control (ctrl) and datapath (alu, operandmuxes) modules.

The first column in Table 3.4 gives the name of the module. The second column gives the number of module level test sequences for which the instructions were generated. The total number of instructions that were generated for the selected module level test sequences is shown in the third column. We fault simulated the instruction sequences at chip level and checked if they detected the fault corresponding to the module level test sequence.

We also found that the technique lends itself well to prioritizing instructions. If the user wants to assign higher priority to certain instructions, then instruction set input to SMV-BMC can be constrained to those instructions. This can be useful if the user wants to avoid instructions like load and store which are multicycle operations.

Table 3.5 gives some examples of instruction sequences that were generated for some of the module level test test sequences from Table 3.4. The first column gives the module name. The second column gives the location of the

Table 3.5: Some example instruction sequences that were generated

Module	Pin path	Fault type	Instruction Sequence	Faults detected
alu	/U2723/C	s-a-1	l.ori r31, r0, 0xffff l.cust5	441
ctrl	/U1091/A	s-a-1	l.ori r31, r0, 0x4 l.ori r30, r0, 0x2 l.nop l.sw 0x0(r31), r31 l.addc r31, r30, r31 l.mfspr r22, r1, 0xf	2424
operandmuxes	/U680/C	s-a-0	l.andi r1, r1, 0xf813 l.movhi r0, 0xf813 l.addi r0, r1, 0xf813 l.sw 0x13(r1), r31 l.movhi r0, 0xfcf3	463

fault within the corresponding module. The type of fault is indicated in the third column. The instruction sequence generated by our mapping technique is shown in the fourth column. These instructions belong to the OR1200's instruction set architecture. The fifth column gives the number of other faults from the base list that were detected by this instruction sequence.

It can be observed from the table that the instruction sequence generated for a specific fault also detected many other faults from the base list. The instruction sequence shown for *ctrl* module detected more than 2000 other faults from the base list. This is because the instruction sequence contained an instruction (l.mfspr) which operated on the special purpose register. This instruction was difficult to generate in the random instruction sequence. How-

ever, our technique generated an instruction sequence containing this instruction. We also found that l.cust5 targeted most of the faults in the *alu* module. This was because l.cust5 instruction does not give complete access to its operands making it is difficult to load it with appropriate values using random instruction sequences.

3.5 Discussion

We have shown a new technique for instruction mapping of module level test sequences. We have used bounded model checking in our technique. In the technique's flow, SMV-BMC can be replaced with a traditional model checker and the technique would still be correct. Mishra et al. [59] provided a way for targeting interesting cases in the pipeline for validation using a traditional model checker. However, they use an abstraction of the processor. Therefore, it is not possible to target any random case since it might not exist in the abstraction. The blowup associated with using traditional model checkers on large designs makes it inefficient to use them without abstracting the design. Bounded model checking has been proposed as a viable alternative to traditional model checking. It provides a partial notion of correctness within the given bound. Although bounded model checking does not guarantee correctness outside the bound, a counterexample obtained during bounded model checking, is definitely valid within that bound. Since our technique translates the module level test sequences into safety properties, a counterexample implies that the safety property does not hold within the specified bound. The

notion of pipeline depth in processors provides a natural way to compute a bound for SMV-BMC.

A major advantage of the technique is that we leverage an existing formal verification engine for solving a problem, that has traditionally been solved by ATPG engines. The existing formal verification engine ensures the automatic and correct operation of the justification like algorithm. Since we add the properties at the RT-Level, any bounded model checking tool can be used for our method. We used SMV-BMC because it was an easily available and robust tool. We can easily substitute it with an engine which works at RT-Level or uses word-level reasoning.

The main overhead of the technique could be viewed as writing the LTL properties. We have automated the generation of controllability properties. There is no manual intervention required to check the controllability of a module level test sequence. In case the processor under test has mechanisms like the *observe only scan chains* [14], we can use this controllability check methodology as a standalone technique to generate tests. Manual intervention is required when the observability property has to be refined. We remove this manual requirement with the introduction of observation abstraction in the following chapter.

Chapter 4

Using observation abstractions for propagation

In Chapter 3, we presented a technique for generating an instruction sequence that produced a given test sequence at a module's inputs. Although the controllability was fully automated in this technique, the propagation of module outputs to processor outputs (observability) had a manual component. In this chapter, we present an automatic technique to propagate module outputs to processor outputs, thereby eliminating the need for manual intervention in the test generation process.

We use Boolean difference for propagation in our technique. Boolean difference has been previously used for test generation [49]. Boolean difference of a function with respect to a signal is *true* if the function depends on the signal. The constraints required for propagation of a module output can be seen as the constraints required to make some primary output dependent on the given module output. Therefore, we express the propagation requirement as a Boolean difference problem. We model the Boolean difference problem as a linear temporal logic (LTL) [54] formula and pass it to a bounded model checker [8]. For our purposes, we will use Boolean difference to model the following property:

There is no dependence between the module output and any of the primary

outputs.

We will then pass this property to a bounded model checker. If this property fails within the given bound then a counterexample is generated, showing that there is a dependence. The constraints needed for propagation are given by the counterexample itself. If the bounded model checker does not produce a counterexample, we can assert that propagation is not possible within the given bound.

Boolean difference, however cannot be expressed succinctly in LTL. Due to the large number of variables in any reasonably sized circuit design, the property (machine) blows up. Hence, an alternate strategy is required to express Boolean difference succinctly in LTL, for our purposes. We circumvent the property blowup problem by a introducing an abstraction at the RT-Level. We call this abstraction as *observation abstraction*. Observation abstractions hide unnecessary details of the design, and retain only the portion of the design relevant to the target LTL property. This enables us to check observability properties modeled as LTL formulas.

4.1 Background

4.1.1 Boolean difference

Boolean difference is an accepted method [49] of representing fault propagation in a circuit. The Boolean difference of any function F with respect to its variable x_i is given by,

$$F(x_1, \cdots, x_{i-1}, 1, x_{i+1}, \cdots, x_n) \oplus F(x_1, \cdots, x_{i-1}, 0, x_{i+1}, \cdots, x_n)$$

The Boolean difference of F with respect to x_i will be *true iff* F depends on x_i. Therefore, Boolean difference can be used to find the dependence of a circuit's output function on any of the circuit's internal signals.

4.2 Automation of propagation

4.2.1 Observability using Boolean difference

A module output m_o is propagated to a primary output P_o *iff* the Boolean difference of P_o with respect to m_o is *true*. Hence, the constraints necessary for propagation would be the constraints that make this Boolean difference *true*. If we *negate* this Boolean difference equation and pass it to SMV-BMC as an LTL formula, along with the design under test, the counterexample produced by SMV-BMC will contain the necessary constraints for propagation.

Boolean difference cannot be expressed succinctly in LTL. When the change in a function with respect to some input variable is expressed in LTL, the resulting formula needs an explicit substitution of the variable in question. If the number of input variables, (in our case, the signals of a circuit) are large, the resulting LTL formula will be intractable. This will result in a space/time blowup of the (bounded) model checker.

In order to overcome this problem, and express Boolean difference succinctly in LTL, we propose an RT-Level abstraction that abstracts, or hides the details that are unnecessary to the LTL formula, thereby making it smaller and more compact. We call this abstraction *observation abstraction*. We outline

the algorithm to construct *observation abstractions* below.

4.2.2 Algorithm

Let M be a Verilog design with k modules m, such that $M = \|_{i=1}^{k} m$ where $\|$ is the parallel composition operator. Let OP be the set of all output signals of M. Let AS be the set of all the statements in M. Consider a module m_i. Let t be a single-bit output signal of m_i, whose (test) value we want to propagate to (at least one of) the outputs. We call this the signal under test.

All the signals that appear on the left hand side of an assignment statement are said to be *defined* and the ones that appear on the right hand side are said to be *used*. For a statement s, let $U(s)$ denote all the signals that are used in s and $D(s)$ denote all the signals that are being defined in s.

Definition 1. *Affectation*

A set of statements S is said to *be affected by* the values of a signal x at a given point if x defines a subset of signals used in S.

Definition 2. *Observability*

A signal x is said to be *observable* if a change in the value of the signal causes a change in the value of at least one of the outputs.

Definition 3. *Observation Abstraction*

An *observation abstraction* w.r.t $\langle t, v \rangle$ where t is the signal under test

and v is a variable, is the transitive closure of all statements that are affected by t, when t is assigned a value v.

We denote this observation abstraction by $A^{t=v}$.

The observation abstraction is meant to provide a means to express the Boolean difference observability property in LTL. The LTL property is now written with the help of this artifact. An algorithm for our technique is given in Figure 1. The observation abstraction is computed with respect to the two values that t can assume, namely 0 and 1. The corresponding observation abstractions, $A^{t=0}$ and $A^{t=1}$ are created such that any statement that is executed when the value of t is 0 is retained in $A^{t=0}$ and any statement that is executed when the value of t is 1 is retained in $A^{t=1}$. This process is done iteratively over all the statements, until a fixpoint is reached. The fixpoint in this case is set of all reachable output signals. This is denoted by $ReachedOutputs$. When no new output signals are reached in the most recent iteration, the fixpoint is assumed to have been reached.

Now, the two observation abstractions, $A^{t=0}$ and $A^{t=1}$ contain all and only those statements that are affected by the values $t = 0$ and $t = 1$. We now write the required LTL property using the observation abstractions. For every observable (output) signal ob, ob^0 and ob^1 are the two "versions" of the signal in the corresponding observation abstraction. If ob^0 is equal to ob^1, it would mean that the signal ob does not change to reflect a change in the value of signal t, and therefore cannot be used to observe t. This is expressed as an LTL property $P_i = G(ob^0 \neq ob^1)$. G is a temporal logic operator that denotes

"over all times", or simply "always". All such properties P_i, written for all output signals can then be ORed together to build property P. The property P can now check if any of the output signals can be used to observe the signal under test.

1. Compute the observation abstraction with respect to $\langle t, 0 \rangle$. Let it be $A^{t=0} = \phi$ initially. Let $ReachedOutputs = \phi$. Let $PrevReachedOutputs = \phi$.
2. Repeat
   (a) $PrevReachedOutputs = ReachedOutputs$
   (b) For every statement s
      i. If s is a statement that is affected by t, and is executed when $t = 0$,
      $A^{t=0} = A^{t=0} \bigcup s$
      ii. If $(D(s) \bigcap OP \neq \phi)$a
      $ReachedOutputs = ReachedOutputs \bigcup (D(s) \bigcap OP)$

   until $ReachedOutputs = PrevReachedOutputs$
3. Compute the observation abstraction with respect to $\langle t, 1 \rangle$. Let it be $A^{t=1}$. (This computation will be similar to the computation of $A^{t=0}$)
4. For every (observed) signal $ob \in OP$, let $ob \in A^{t=0}$ be called ob^0 and $ob \in A^{t=1}$ be called ob^1. Write an LTL property $P_i = G(ob^0 \neq ob^1)$. Let $P = P_0 \mid P_1 \ldots \mid P_n$ where n is the total number of observed outputs. P is the required LTL property.

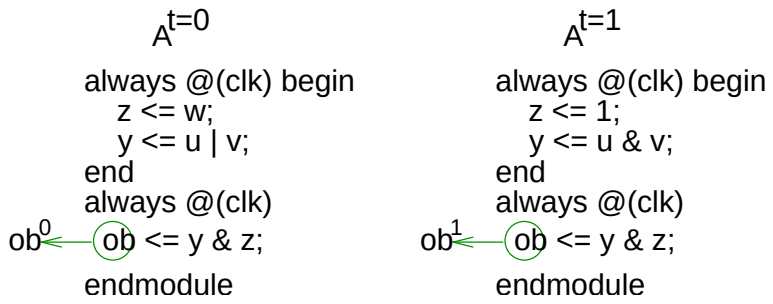
Figure 4.1: Algorithm for our technique using observation abstractions for writing LTL properties.

An example illustrating the working of the algorithm is shown in Figures 4.2 and 4.3.

The argument for the soundness of the algorithm can be established

```
module example(clk,t,w,x,y,z,op);
input clk,t,u,v,w,x,y,z;
output ob;
always @(clk) begin
   z <= t | w;
   if (t)
      y <= u & v;
   else
      y <= u | v;
   end
always @(clk)
   ob <= y & z;
endmodule
```

Figure 4.2: Sample Verilog model. Here t (output of some other module) is the signal that has to be propagated to an output of the current module. *ob* is the only output signal.



(a) Observation abstraction with respect to $\langle t, 0\rangle$. ob^0 is the version of *ob* in $A^{t=0}$.

(b) Observation abstraction with respect to $\langle t, 1\rangle$. ob^1 is the version of *ob* in $A^{t=1}$.

Figure 4.3: An example illustrating the algorithm. The property P for this example is $G(ob^1 \neq ob^0)$. This reduces to $G(((\text{u} \mid \text{v}) \ \& \ \text{w}) \neq (\ \text{u} \ \& \ \text{v}))$.

by a proof by contradiction. For the LTL property to fail portraying the signal propagation on observation abstractions $A^{t=0}$ and $A^{t=1}$, there must be at least one statement that can be reached in the original Verilog program, that cannot be reached in $A^{t=0} \bigcup A^{t=1}$. However, by definition, $A^{t=0}$ contains all the statements reachable when $t = 0$ and $A^{t=1}$ contains all the statements reachable when $t = 1$. Since these are the only two values allowed for a single bit Boolean variable t, it can be shown that the desired LTL property for propagation can be expressed using observation abstractions for a signal. A similar argument can be presented for the reverse direction establishing the completeness of the observation abstractions.

The algorithm for computing observation abstractions is linear in the size of the program. Therefore, the scalability of the technique is not challenged by the size of the program or the number of observed variables. However, the capacity of the lower level engine (in this case a bounded model checking engine) is limited, and depending on the number and size of the variables, there might be a blowup, despite the reduction in size afforded by these abstractions.

4.2.3 Constraints extraction

Once P is defined, we obtain the observability property O by negating P. We pass O to SMV-BMC along with the model $A^{t=0} \bigcup A^{t=1}$.

The negation of the property P, or O, implies that there exists no signal in OP, that can be used to observe the signal under test t. If the property

does not hold true, SMV-BMC will provide a counterexample, indicating that there exists at least one output signal, that can be used to observe t. The counterexample to the property O is a witness to the property P. Hence, the counterexample will contain the constraints necessary to propagate the signal under test, t.

4.2.4 Functional test generation

The controllability part for functional test generation is similar to the one we described in Chapter 3. The final property is of the form $(C \implies O)$, where C is the controllability property and O is the observability property derived in previous sections. We transform the given test sequence at the module inputs to an LTL property to obtain the controllability property. We pass the final property to SMV-BMC. We also add constraints so that external stalls are disabled. We constrain the input space searched by SMV-BMC to the processor's ISA so that counterexample produced by SMV-BMC is possible through valid instructions. We add the design in the backward cone of influence of the module inputs and the observation abstractions to get the model to pass to SMV-BMC. If SMV-BMC produces a counterexample on this model under the constraints that we have derived then the counterexample will contain the instruction sequence necessary to produce the module test sequence at the module inputs as well as propagate the module output to primary outputs. SMV-BMC has to be given a bound for it to work. We provide a bound which is the maximum number of cycles an instruction can

be present in the pipeline. If SMV-BMC does not produce a counterexample, then we term the sequence to be *unmappable*. Those sequences are rejected. Our results of rejected test sequences is valid within the bound.

4.3 Experimental results

Our tool to generate observation abstractions was written in C++ over a C-based Verilog parser. We then performed our experiments on the OR1200 processor [89]. Since the primary target of our technique are again the hard-to-detect faults, we used the same instruction sequence generated for experiments in Chapter 3 and fault-graded the OR1200 CPU core for all possible stuck-at faults using a commercially available tool [20]. The fault coverage saturated around 68%. As in Chapter 3, the list of faults that were left undetected formed the base list. We applied our technique on this base list. This base list represents the hard-to-detect faults of OR1200.

Table 4.1 provides an overview of the results that were obtained from the experiments. We will now explain the various experiments that were performed and the results that were obtained.

4.3.1 Fully automated functional test generation

We sorted the base list (obtained above) based on modules. We generated a module level test sequence, using a commercially available ATPG tool [20], for each fault individually. We then used our technique for functional test generation (explained in Section 4.2.4) to map these module level

Table 4.1: An overview of the results obtained

Overview of results	
No. of instructions in OR1200 ISA	92
No. of combinational library primitives in the synthesized design	19621
No. of sequential library primitives in the synthesized design	1628
Total no. of uncollapsed faults	82098
Fault coverage of the random instruction sequence	68%
No. of faults in base list	26423
No. of additional faults detected	11296
Overall fault coverage	82%

test sequences to instruction sequences. Table 4.2 shows the results that were obtained for this experiment.

The first column in Table 4.2 gives the name of each module. Second column gives the number of faults of each module present in the base list. The number of faults for which a *good* sequence was generated by the commercial ATPG tool is given in the third column. A sequence is *good iff* it provides both:

- The required values needed at the module inputs for controllability
- The module output signal through which the fault can be propagated further

We note from the results that for some of the modules ('rf', 'freeze' and 'except'), the ATPG tool did not produce a *good* sequence for a majority of the faults.

Table 4.2: Result for instruction mapping of module level test sequences

Module	Module faults	Total tests	Mapped tests	Rej. tests	Timed out	Effi. E (%)	Av. abs. time(s)	Av. mapping time(s)
ctrl	1981	1764	159	1434	171	90.3	1.0	24.4
if	1764	1420	64	1335	21	98.5	1.5	34.5
lsu	1332	1245	291	91	861	30.8	1.1	28.1
wbmux	894	857	637	97	123	85.6	1.5	47.3
alu	2315	2311	38	901	1372	40.6	1.3	9.1
except	4138	474	41	261	172	63.7	0.9	18.4
genpc	2612	2352	1089	1239	24	98.9	0.3	12.9
rf	5148	1801	5	1668	128	92.8	1.3	28.0
sprs	4910	4851	229	2482	2140	55.8	0.7	17.5
freeze	73	27	1	24	2	92.5	1.1	23.0
opmux	362	217	36	176	5	97.6	1.1	26.7

We used SMV-BMC tool for mapping the *good* sequences to instruction sequences. We used a bound of 7 for these experiments. The fourth column of Table 4.2 gives the number of sequences that were successfully mapped to instructions. The fifth column gives the number of sequences that were rejected since they were not mappable within this bound, *i.e.*, no counterexample was found within the bound of 7. SMV-BMC tool was liable to explode in state space for some cases. Hence, we placed a time-out, of 90 seconds, to limit the amount of effort that was spent on mapping a *good* sequence. The sixth column shows the number of sequences for which the SMV-BMC tool timed-out. If the experiment had been performed with out a time-out, some more of the sequences could have produced a result. We calculate the mapping

efficiency (E) as follows:

$$E = \frac{M + R}{T} * 100$$

where

M – No. of mapped sequences,

R – No. of rejected sequences,

T – Total no. of sequences.

We note from the table that the efficiency is above 90% for most of the modules. However, the overall mapping efficiency is 71%. This is because the efficiency falls below 50% for 'alu' and 'lsu' modules. As shown in the experiments of Section 4.3.3, the time-outs which contribute towards the lowering of efficiency happen due the observability phase of the technique. The average time taken for producing the abstraction and for the mapping process is shown in the last two columns. The average time taken for the abstraction process is for all the *good* sequences. As can be seen from the table the abstractions were generally generated in one second. The average time taken for mapping process was calculated only for those sequences for which the SMV-BMC produced a result, (*i.e.*, SMV-BMC did not time-out).

4.3.2 Controllability check

We did an experiment to perform a controllability check that helps interpret the results obtained in Table 4.2 better. We used the same *good* sequences from the experiment explained above and performed only the controllability check, *i.e.*, we did not give the observability property to SMV-BMC.

We used the same bound and time-out limits as the experiment in Section 4.3.1. Table 4.3 shows the results that were obtained for this experiment. The total number of sequences for each module is shown in the second column. The third column shows the number of test sequences for each module that were found to be controllable functionally. The fourth column shows the number of test sequences that were found to be uncontrollable using instructions, *i.e.*, no counterexample was found within the given bound. SMV-BMC produced a result for all the *good* sequences, *i.e.*, there were no time-outs.

Table 4.3: Result for controllability check

Module	Total test sequences	Controllable test sequences	Rejected test sequences
ctrl	1764	1036	728
if	1420	1174	246
lsu	1245	1154	91
wbmux	857	825	32
alu	2311	1410	901
except	474	221	253
genpc	2352	1145	1207
rf	1801	1061	740
freeze	27	3	24
operandmuxes	217	170	47
sprs	4851	2438	2413

4.3.3 Observability check

Since there were no time-outs in the controllability check of Section 4.3.2, the observability factor must be the one that increases the complexity of the

state-space search and causes SMV-BMC to produce no result for some sequences in the experiments of Section 4.3.1. We performed an observability check on *good* test sequences to provide an insight into those cases. For this experiment we passed only the observability property to the SMV-BMC tool. We used the same bound (of 7) and time-out (of 90 seconds) as the previous experiments. The results obtained are shown in Table 4.4.

Table 4.4: Result for time-out experiment on observability phase

Module	Total no. of signals	No. of time-outs	Projected time-outs	Time-out ratio
ctrl	130	38	392	10.3
if	69	23	472	20.5
lsu	61	32	996	31.1
wbmux	65	33	196	5.9
alu	35	3	1493	497.7
except	87	45	268	5.9
genpc	34	1	56	56.0
rf	96	70	1293	18.5
freeze	4	2	7	3.5
opmux	85	21	21	1.0
sprs	163	63	2904	46.1

We started with the same *good* sequences. We made a list of module output signals that the *good* sequences required to be propagated. As mentioned in Section 4.3.1, each of these sequences specified a module output signal through which the effect of the fault can be further propagated. Many patterns identified same output signals for propagation. We performed the experiment once for each module output signals that was identified. The number

of different module output signals that were identified by *good* sequences for each module is given in the second column of Table 4.4. The third column gives the number of signals for which the experiment did not produce a result. The fourth column gives the number of patterns that used a signal whose observability check timed-out to be propagated, this represents the number of time-outs that can potentially occur while performing functional test generation of Section 4.3.1. We note that numbers in this column are higher than the numbers in the time-out column of Table 4.2. This is due to either of the following.

- While performing instruction mapping for a pattern, satisfaction of the controllability property probably places a higher constraint on the state space that needs to be searched. This presents a reduced state space to search while proving (or disproving) the observability property.
- Some patterns whose propagation requirement timed-out in Table 4.4 were rejected (in the experiments of Section 4.3.1) because they were either functionally uncontrollable or their propagation requirement contradicted with the controllability requirement.

The final column of Table 4.4 gives the time-out ratio. Time-out ratio is the ratio of numbers in fourth column to the numbers in the third column. We can note from this column that modules (like 'alu', 'sprs' and 'lsu') whose mapping efficiency were found to be low in Table 4.2, have a rather large time-out ratio. This implies that module level test sequences for these modules

require the same output signal (whose observability check timed-out) to be propagated a larger number of times than other modules. This leads to a lowering of efficiency. A method for increasing the efficiency will be to force the ATPG tool to identify some other output signal (which did not time-out) for propagation while generating the tests at the module level. Since we did not have enough control over the ATPG tool presently, we could not try this successfully.

One result of the observability check not shown in the Table 4.4 is that except for the 'ctrl' module, in every other module there were no module output signals that failed the observability requirement. This is expected, since in a processor signals that cannot be propagated under ISA constraints mostly represent a redundant part of the design. In fact, the signals of the 'ctrl' which could not be propagated were found to be redundant.

4.3.4 Fault coverage

We derived test sequences at the OR1200 inputs from the counterexamples that were generated by SMV-BMC. For each module, we concatenated these sequences and fault graded OR1200 for the concatenated sequence. The results for this experiment are shown in Table 4.5.

The second column in the table shows the number of faults from the base list, (*i.e.*, the faults undetected by the random instruction sequence) that were detected by the concatenated test sequence derived for each module. There was an overlap in the additional faults detected by the test sequences

Table 4.5: Fault coverage results

Module	Additional faults detected
ctrl	9011
if	2252
lsu	4642
wbmux	9280
alu	3454
except	4120
genpc	7147
rf	4501
freeze	1798
opmux	6008
sprs	7904

of each module. Overall, 11296 faults from base list were detected. This represented a 14% increase in fault coverage. Hence, the overall fault coverage was 82%. It has to be noted here that this number is for the total fault coverage. We did not eliminate the functionally untestable faults. Chen et al. [17] observed that a 74% total fault coverage of a block (a combinational block since that method requires block under test to be combinational) represents a 90% coverage of functionally testable faults. Therefore, we speculate that 82% overall fault coverage represents a coverage of above 90% of functionally testable faults. This result is also encouraging because [55] shows that a 80% coverage with functional tests identifies more defective chips that structural tests with far higher coverage. Moreover, the tests we generate can be applied at-speed. The experiments done by McCluskey et al. show that at-speed tests

catch more defective chips than same tests applied at a slower speed.

4.4 Discussion

We have a proposed a new technique that fully automates the process of functional test generation targeting specific faults. We achieved it by proposing a new technique for propagating module test outputs to primary outputs using the ISA. This technique is based on an abstraction at RT-Level. We have given the algorithm for abstraction. We have also shown the effectiveness of our technique on an off-the-shelf processor.

A major contribution of this chapter is the observation abstraction. This abstraction mechanism allows us to represent Boolean difference succinctly in LTL. The observation abstraction also allows us to express the propagation requirement at the RT-Level. We can substitute SMV's BMC engine in our technique with any verification engine which takes in RT-Level input. This allows us to use tools which use word-level reasoning in the future.

The mapping process of the technique is removed from the ATPG that produces the sequence for testing a fault at module level. This is a drawback since there might be more than one sequence that can detect a fault. Overcoming this problem by using a feedback to the ATPG engine producing the module level test sequence are part of our further improvements detailed in subsequent chapters.

Chapter 5

Generating instructions targeting delay defects

New kinds of defects are becoming more common with the emerging manufacturing methodologies [25]. Some of these defects are resistive, *i.e.*, the nodes that have these defects are neither open nor shorted, but have a change in their resistance. Other defects are symptoms of changes in the performance caused by the variations in the process technology [11]. The common factor with both these kinds of defects is that they manifest themselves as changes in the delay of the circuit under test. Therefore, they are called *delay defects*. Many delay defects may affect only the longer paths in the circuit because the change in delay may be small. Therefore, they might escape detection from traditional testing techniques.

We used the stuck-at fault model as our target in the previous chapters. As the stuck-at fault model is not sufficient for delay defects, many delay fault models ([75], [15], [43]) have been proposed. The common aspect in all these models is that they test for a transition through a path rather than a single value at a node. Therefore, tests for these fault models contain at least two vectors, necessitating a change in the scan technique. This led to *broadside testing* [69], where ATPG spreads to two cycles of logic, and the

path under test is in the second cycle of logic. Nodes in the first cycle of logic are used to produce the second vector needed to test the path. The need for such constraints was overcome using *enhanced scan* [21]. Each enhanced scan element contains two flops. However, research [47] shows that a large portion of structurally testable delay faults (*i.e.*, faults that are testable in presence of scan) are actually functionally unsensitizable. Hence, using scan for delay testing can lead to considerable yield loss due to overtesting. Moreover, industrial results [25] show that at-speed functional tests are far more effective in catching chips with delay defects. Therefore, there is a need to *functionally* test chips for delay defects. We provide a technique to do so in this chapter.

Our technique consists of three main parts: a) a local Delay annotated (PODEM based) Automatic Test Pattern Generator (DATPG) b) a functional mapping procedure and c) an intelligent feedback mechanism. The fault model we use is the improved unified fault model (IUFM) [43]. The faults in this fault model, like any good small delay fault model, are actually paths. We use DATPG to generate the *locally true* paths We read in the standard delay format (SDF) files provided by the synthesis tool and annotate the synthesized netlist with corresponding delays. DATPG provides the paths that have a delay above a given threshold and are locally true. To ensure that defects are not masked, we use the pseudo-robustness constraints. The paths along with the pseudo-robustness constraints are given to the functional mapping procedure to generate the instruction sequence (if one exists) that can test the path. We apply the functional mapping procedure at the gate-level here.

We do so because we found that in case of delay faults the tests generated are more unlikely to be mapped. If the functional mapping procedure finds the path is not pseudo-robustly testable then the feedback mechanism uses an intelligent heuristic which takes advantage of the way DATPG generates paths, to identify the sub-path which causes the path to be functionally untestable and stores it. Any path which has this non-functional sub-path is discarded quickly in the future. The results that we provide for non-functionality of paths mean that those paths are not detectable with pseudo-robust conditions under functional constraints within the given bound.

The contributions of this chapter are outlined below.

- We provide a *fully automatic* technique to generate instruction sequences to pseudo-robustly test delay defects in processors.
- We read in standard formats (part of existing design flow); therefore, this technique can be easily ported to any processor.
- We show that our technique can obtain 96% fault coverage efficiency on the OR1200 processor design.

5.1 Background

5.1.1 Fault models

A good fault model is essential for any testing methodology. The stuck-at fault model cannot adequately portray the effects of delay defects. Delay defects are generally modeled by fault models like transition fault model [86],

gate delay fault model [15] or path delay fault model [75]. Test generation using transition fault model involves generating vectors for first initializing the net under test to a value and then producing a transition on the same net. Test vectors for this model will have a good coverage for gross delay defects since nets with gross delay defect will fail to produce the required transition. However, the paths through the nets that are activated by the vectors might not be long (slow) enough to cause a failure in the presence of a small delay defect. Therefore, the circuits with small delay defects might escape detection. The gate delay fault model improves on this by targeting only the longest path through each net. However, this model fails to cover the cumulative effect of small delay defects distributed over several gates. The path delay fault model targets those kinds of defects. Test generation using this model involves generating tests for all the paths of the circuit under test. However, the number of paths in a circuit can grow exponentially with the number of nets in a circuit. Therefore, it is tough to generate vectors for all the faults in this fault model. A new fault model, called improved unified fault model (IUFM), was proposed in [43]. It involves generating tests for the longest sensitizable path (both rising and falling) passing through all the nets and also all the sensitizable paths above a certain threshold in the circuit. The number of longest paths through each net is linear in terms of the number of nets in the circuit. Moreover, by choosing an intelligent threshold, it is possible to represent the effects of distributed small delay defects and also test for only a moderate number of paths. Therefore, it is easier to achieve higher coverage

using this fault model. Hence, this fault model provides a good framework to target small delay defects.

5.1.2 Delay tests

The tests for paths should have at least two vectors (V_1,V_2). The vector V_1 produces the necessary first cycle value at each net in the path as defined by the transitions on the nets. In case of a delay defect, the transition is delayed or absent in the chip, V_2 propagates this information through the path to the end of the path. The delay tests are classified into various ways as described in [44]. Before explaining these classification we need to define a few terms.

Node: A node is a pair of a gate output and a transition (rising or falling). Each node has a delay associated with it, as given by the SDF file.

Path: A path is a sequence of nodes and it has a delay associated with it. The delay of a path is the sum of the delays of nodes in the path. The sequence starts at a pseudo-input[1] and ends at a pseudo-output[2]. There are no memory elements on the path. The output of each gate in the path is connected to an input of the succeeding gate in the sequence. The pseudo-input at the beginning of the path is connected to an input of the first gate in the sequence and the last gate in the sequence is connected to the pseudo-output.

Sub-path: Any sub-sequence of a path.

True path: A path is *true* if there is a test for it. A path can be *locally* or

[1]A primary input or an output of a memory element
[2]A primary output or an input to a memory element

globally true, depending on whether a combinational or a sequential test exists for it. In our case, a sequential test is a set of instruction sequences and the wrong value must be propagated from the flop at the end of the path to an observable point in processor (primary output, register file etc.).

On-input – A signal is on-input of path P if it is on P.

Off-input – A signal is off-input of path P if it is an input to a gate in P and it is not an on-input. We also use the term side-input in this dissertation to mean off-input.

Robust off-input – An off-input i of a gate on path P is a robust off-input if either of the following conditions is satisfied (here j is the on-input to the same gate):

1. If j transitions from a controlling value to a non-controlling value, i should have the same transition or should have a stable (glitch free) non-controlling value in both cycles of the test

2. If j transitions from a non-controlling value to a controlling value, i should have a stable (glitch free) non-controlling value in the both cycles of the test.

Non-robust off-input – An off-input of a gate on path P is a non-robust off-input if it has a transition from controlling value to a non-controlling value when the on-input has a transition from non-controlling value to a controlling value. For example, in case of an AND gate, if the off-input is rising when the on-input is falling, the off-input is a non-robust off-input. Note that, in this

case, the transition will be propagated only when the off-input arrives earlier than the on-input. Therefore, timing information will be necessary to make sure that the test is valid.

Now we can explain the classifications of delay tests.

Robust tests: If a delay test for a path has all the values to make sure that the transition is propagated through the path and also ensures that all the off-inputs are robust off-inputs, then it is *robust*. A robust delay test for a path can guarantee that a wrong value will be latched on to the memory element at the end of the path when the path is too slow, even when there are other delay faults in the circuit.

Non-robust tests: If a delay test for a path has all the values to make sure that the transition is propagated through the path and at least one of the off-inputs is a non-robust off-input, then it is *non-robust*. A *non-robust* test can guarantee the detection of a delay fault when no other delay fault is present.

It will be ideal if each path under test is robustly tested. However, it is not practically possible to achieve such a goal. On the other hand, fault coverage numbers using non-robust tests are not highly trustworthy. A robustness condition in between these two would be a good solution. Such a condition, called *pseudo-robust*, has gained some recent traction [83], [6]. Pseudo-robust tests have robustness conditions similar to that of robust tests. However, the requirement of off-inputs being glitch free is dropped. In Figure 5.1, an example

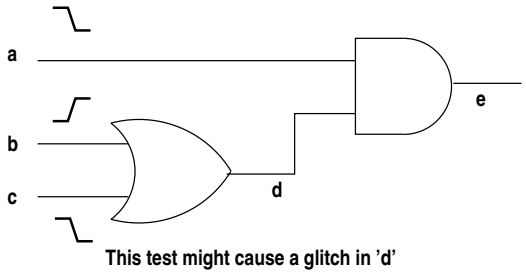


Figure 5.1: A test which is pseudo-robust but not robust

test is given which is not robust but pseudo-robust because the signal value in **d** can have a glitch. In this dissertation, we will deal with pseudo-robust excitation in case of delay tests.

5.2 Technique

Our technique for delay test generation has two phases since IUFM is used. In the first phase, all the paths above a preset threshold are functionally mapped. All the nodes in the mapped paths are said to be covered. In the second, the longest path through each uncovered node is mapped.

The block diagram of the technique is given in Figure 5.2. As shown in the figure, we start with the synthesized netlist and the SDF file. These are the inputs to the preprocessing step which annotates each net in the netlist with the delays (for all transitions at the net) as given by the SDF file. This step also determines the threshold to be given to the DATPG.

In the first phase of test generation, DATPG generates all the locally *true* paths above the threshold (set to a value which is a function of clock

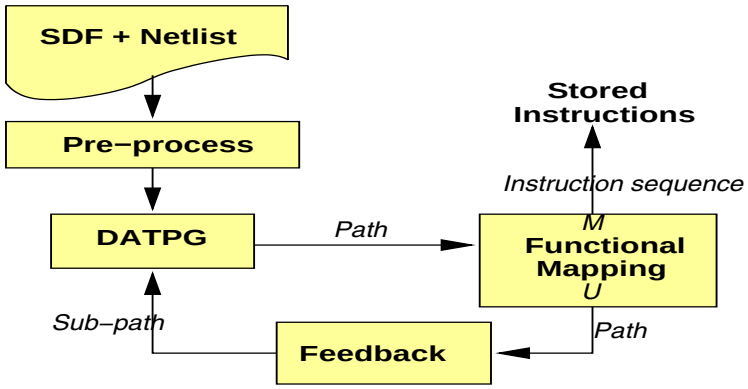


Figure 5.2: Block diagram of the technique

period) and calls the mapping procedure to generate instruction sequences that detect those paths. In the second phase, DATPG is given a threshold and a modified circuit such that all the *locally true* paths above the threshold contain the node being targeted. DATPG generates those paths and dynamically updates the threshold with a path's delay if that path is found to be functionally mappable. In both phases, if a path is found to be functionally unmappable, the feedback mechanism finds a sub-path that is non-functional. This non-functional sub-path is stored so that, in the future, all the paths that contain this sub-path are not given to the functional mapping procedure. We will now explain the main blocks of this technique.

5.2.1 DATPG

Before going into the details of DATPG, we will elaborate on the need for such a procedure. Traditionally, delay testing for path delay faults contains two parts, a) path generation using static timing analysis (STA) methods and b) generating tests for those paths. However, there are some philosophical and

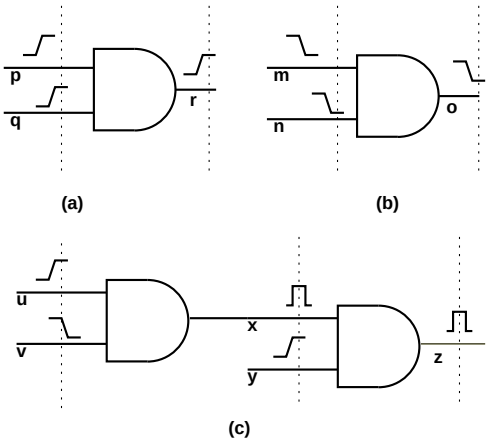


Figure 5.3: Illustration of difference between STA and delay testing.

engineering issues with using STA tools in delay testing.

- STA and delay testing need different kinds of paths. In Figure 5.3(a), the AND gate has two rising transitions at the input. STA will choose only one path, **qr**, since **q** arrives later than **p** and hence causes the transition in **r**. However, for pseudo-robust tests, both the paths **qr** and **pr** must be considered, since if a delay fault causes either transition to be absent, the transition in the output will be absent. Figure 5.3(b) shows another case. In this case neither path **mo** nor path **no** are pseudo-robust. However, STA will provide the path **mo**.

- STA is meant to be used to derive a conservative estimate of the clock period. Therefore, paths due to glitches will be considered. For example, let us suppose a test causes the transitions illustrated in Figure 5.3(c). STA will consider this as a test for path **yz**. However, the path **yz** has a

transition in it only if a glitch occurs in **x** because of the late arrival of **v**. Let us suppose this path and vector are chosen for delay test. Now, if due to process variations on the manufactured chip, signal **v** arrives earlier, there is no glitch on **x** and it is a stable zero. This will imply that there is no transition on output **z**. This is the correct operation intended by the designer for the given inputs. However, our delay test will fail the chip since the glitch did not occur. Therefore, in the presence of process variations, considering glitch based paths for delay testing might lead to problems.

- Using a STA tool would mean that the test generation will be separate from the path selection. This can lead to loss of coverage. Moreover, most STA tools are not optimized to find *locally true* paths.

Our test generator is based on PODEM and uses a delay based heuristic for propagation [39]. As described in section 5.1.2, delay tests have two vectors. The first vector sets the required value in the first clock cycle and the second vector is for propagation. If we have rising (falling) transition, the second vector has conditions for propagation of a stuck-at-0 (stuck-at-1) fault. If the transitions are always at the input, a single cycle PODEM algorithm simulates the delay test generation algorithm. For targeting transitions at an internal node, we increase the delay of that node so that all the long paths contain that node. The flowchart of DATPG is shown in Figure 5.4. This flowchart does not show the feedback from the mapping procedure. The terms backtrace,

backtrack, D-frontier and x-path hold the same meaning as in the traditional PODEM algorithm. The shaded boxes show the areas where the algorithm differs from traditional PODEM.

The DATPG procedure deals with the circuit as a combinational design and backtraces (implies) only up to flop outputs (inputs). Before we start the ATPG procedure, we do a computation to figure at each node, the maximum possible delay of a path through that node. This is a linear time computation. Each node has a source and a sink delay. The source delay of a (pseudo) input is zero and sink delay of a (pseudo) output is zero. The source and sink delays of the rest of the nodes are computed iteratively. The source delay of a node is the sum of maximum of source delays of all fanin nodes and the propagation delay of the node. Similar computation is done for sink delays using fanouts of each gate. The maximum possible delay of a path containing a node is a function of its source, sink and propagation delays. We use these delay values during ATPG as described below.

As shown in Figure 5.4, we keep inserting transitions (stuck-at faults) at the input and propagate them. The choice of input/transition is made based on decreasing order of the sink delays of the combination. After implication, if we find that no output has a transition, we know that we have to propagate the transition through some D-frontier. If no D-frontier is present, that means the transition is not propagatable any more and we need to backtrack. However, if a D-frontier is present, we choose the D-frontier for propagation based on the source/sink delay heuristic. The backtrace is done in a similar way to

general PODEM algorithms. If we find that a transition is propagated to an output, we try to find all the paths that have been simulated[3]. If such a path's delay is greater than the given threshold, it is checked against the non-functional sub-path database to find if it contains any of them. If it does not contain any non-functional sub-path, it is given to the functional mapping procedure. A change from traditional PODEM is that we do not stop when an input transition has been propagated once; we continue till all possible paths starting with that input transition longer than the threshold delay are found. The updating of threshold is not shown in the flowchart. In the second phase of test generation, the threshold is updated with the delay of a path when that path is found to be functional, so that we can target the longest paths through the node.

5.2.2 Functional mapping procedure

Once it is ascertained that a path above threshold does not contain any non-functional sub-paths, we invoke the functional mapping procedure to map it to instructions. The procedure is similar to the one described in previous chapters with some changes. The main difference is that the mapping uses the gate-level model now. We generate a similar LTL property of the form $(C \implies O)$. C in the property contains the path controllability constraints, pseudo-robustness constraints and instruction constraints. O includes

[3] All the sequences of gates starting at the input and ending at any output with transitions at the outputs of all intermediate gates are said to be simulated

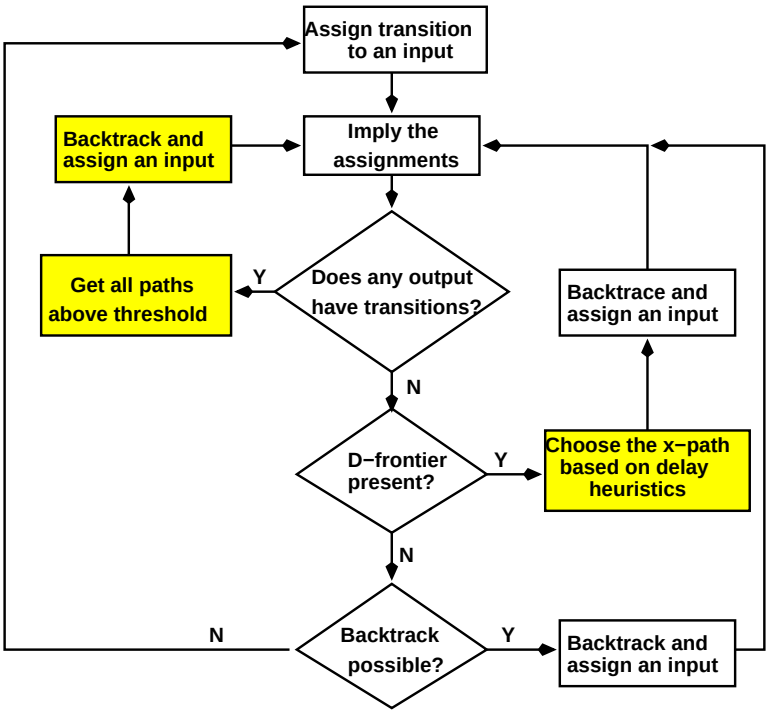


Figure 5.4: Flowchart of the PODEM based delay test generation technique

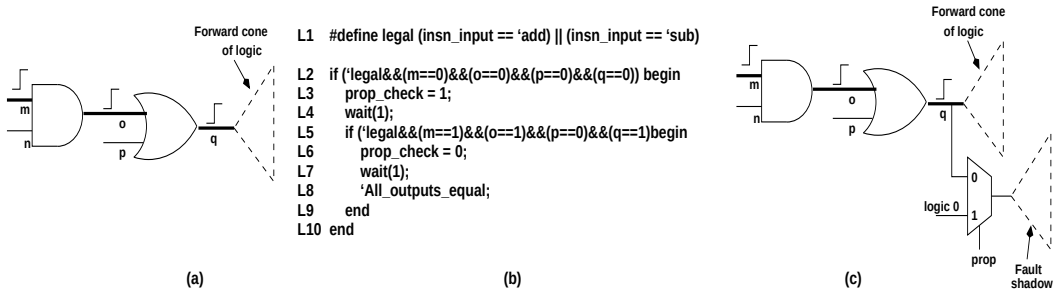


Figure 5.5: Illustration of functional mapping

the observability constraints. An example of the constraint extraction is given Figure 5.5. Figure 5.5(a) shows a path produced by DATPG, Figure 5.5(b) shows the property generated to map the path and Figure 5.5(c) shows the modification done to the design given to SMV-BMC for observability.

5.2.2.1 Generating antecedent

C, the antecedent of the final property contains the path controllability, pseudo-robustness constraints and instruction constraints. We derive the controllability constraints directly from the path. For example, let us suppose the path given by DATPG looks like Figure 5.5(a). The controllability constraints for the path are: *m rising, o rising, q rising.* This is built into the *if* conditions of the property as shown in lines L2 and L5 of the property in Figure 5.5(b). The *wait* statement of line L4 specifies that condition in L5 should happen one cycle after the condition in line L2. It can be seen that the conditions for m,o and q are as required. Now we need to generate the pseudo-robustness constraints. This requires additional specification if any gate in the path has a transition ending in a controlling value. In the example, rising input of the

OR gate is such a transition. Therefore, we must specify that other input must have non-controlling values. This is why p is specified to stable '0' value in the conditions of lines L2 and L5. We also add the instruction constraints. This is done by making sure at every cycle instruction input is a legal instruction. This is illustrated in Figure 5.5(b), if there are only two legal instructions ('*add* and '*sub*). Now we have C of the final property.

5.2.2.2 Boolean difference for observability

We use Boolean difference to express the observability constraints. We transform the design using the fault shadow methodology described in [49]. In this method all the logic in the forward cone of faulty net are duplicated. In our case the forward cone of the last node in the path will be duplicated. This is shown in Figure 5.5(c). In [49], the faulty value is *always* fed to one cone and the correct value is fed to another. Conditions to make outputs of these cones different are then found. However, we deal with sequential logic and in our case fault is triggered (when the path is excited) in only one cycle (the second cycle of the transitions in path). Therefore, we introduce a multiplexer which propagates the faulty value in only one cycle and in the other cycles propagates the value evaluated from the circuit. For this purpose, new signals *prop* and *prop_check* are introduced. We now add an assertion that the outputs of correct and fault shadow logic are always equal (line L8). This assertion is the consequent O of the property .

5.2.2.3 Using a bounded model checker

We pass the property generated in Sections 5.2.2.1 and 5.2.2.2 to a bounded model checker (SMV-BMC in our case) along with the transformed (as explained in Section 5.2.2.2) design. We give a bound to the bounded model checker based on the maximum number of cycles an instruction can be present in the pipeline of the processor in the absence of stalls. A counterexample, if produced by the bounded model checker, will simulate the path because it will satisfy C. Therefore, path controllability and robustness constraints will be satisfied. The assertion in O will be falsified; therefore, the value of some output signal is different in the faulty and correct versions. Therefore, the path effect is propagated. As explained in the previous chapters, the instructional constraints in C will make sure that the counterexample is possible through legal instructions. The instruction sequence for detecting the path is present in the counterexample, which is easily extracted.

5.2.3 Feedback

The technique has a feedback loop from the mapping procedure to DATPG. Without the feedback loop DATPG might repeatedly provide very similar functionally untestable paths. The time taken to map a path to instructions is far higher than the time taken to generate the path using DATPG. Therefore, any time a path can be rejected as functionally untestable without actually mapping it, much time and effort can be saved. We make use of the fact that:

$$A\ sub-path\ of\ path\ P\ is\ functionally\ uncontrollable$$

$$\Longrightarrow$$

$$Path\ P\ is\ functionally\ uncontrollable$$

This is because a path can be seen as a conjunction of a set of predicates on gate outputs. Any sub-path of the path is a conjunction of a sub-set of predicates. Therefore, the sub-path is a weaker conjunction. If a weaker conjunction is unsatisfiable under a set of environmental conditions[4], then the stronger conjunction is also unsatisfiable. Therefore, a path is functionally uncontrollable if any sub-path of it is functionally uncontrollable.

It will be advantageous to maintain a database of sub-paths that are functionally uncontrollable. This will necessitate a procedure to find the non-functional sub-paths of a functionally uncontrollable path. There are $\frac{n(n+1)}{2}$ possible sub-paths for a path of n nodes. We do not want to check if each of those sub-paths is functionally mappable, since it can cause the time taken to quadratically increase. Therefore, we propose an intelligent heuristic. This heuristic takes advantage of the way paths are generated by DATPG. As explained in Section 5.2.1, each run of DATPG starts with a transition at an input and finds all possible ways that transition can be propagated to outputs. Therefore, the initial parts of the consecutive paths are likely to be the same. We try to find a sub-path starting at the input that is functionally uncontrollable. However, there are n possible such sub-paths and trying the mapping process on all of them can also be time consuming for large designs

[4]Constraints derived from the ISA and functional environment

where each path can have more than 50-100 nodes. To avoid this, we do a logarithmic reduction where we start with the first half of the path and recursively test the controllability of the first half of the sub-path. We stop as soon as we reach the sub-path that is functionally controllable. The pseudo-code for this procedure is given below.

```
Find_subpath(path,N) {
  //Get sub-path starting at input half
  //the size of current sub-path
  subpath = getsubpath(N/2);
  if (functionally_controllable(subpath) {
      return N;
  } else {
    Find_subpath(N/2);
  }
}
```

The initial parameters to this function are the original path and n. This procedure will result in only $log(n)$ functional controllability checks. Note that we do only functional *controllability* checks in this feedback procedure. This controllability check is similar to the mapping described in Section 5.2.2. The only difference is that we do not provide the observability constraints to the mapping procedure, *i.e.*, we make O (the consequent) trivially falsifiable in the property given to the bounded model checker. This procedure may not

return the smallest possible sub-path starting at input that is functionally uncontrollable. However, that is a trade-off if we want to limit the number of functional controllability checks. However, since the initial halving of the paths might discard lot of nodes, we add an additional check. If the initial half-path is functionally controllable then we intersect the current path with the previous untestable path and check for its functional controllability.

5.3 Experimental results

We performed our experiments again on the OR1200 processor. We synthesized OR1200 using a 0.18u technology available from TSMC. Some details about the synthesized OR1200 processor core are provided in Table 5.1.

Table 5.1: An overview of the synthesized OR1200 design

No. of instructions in OR1200 ISA	92
No. of combinational gates in the synthesized design	15878
No. of sequential elements in the synthesized design	1594

Bound that we used for our experiments is 7. We used a threshold of 80% of the clock cycle for the first phase of testing. The results of first phase are given in Table 5.2. The first column shows the number of paths that were found to be above the threshold by DATPG. As explained in Section 5.2.1, DATPG tries to propagate all possible input/transition combinations using PODEM.

Table 5.2: Results of Phase 1 of experiment

No. of Paths	Drop	Functionally Testable	Functionally Redundant	Time out
27424	12	15118	12106	200

The second column shows the number of such combinations for which DATPG reached a backtrack limit of 10000. 15118 paths of the 27424 were found to be functionally testable, as shown in third column. 12106 of the paths were found to be functionally untestable (hence functionally redundant[5]) and the mapping procedure timed out for 200 of the paths (timeout limit used was 150 seconds). Those results are depicted in the fourth and fifth columns of Table 5.2.

In the second phase, we generated the longest functionally testable path though each node not covered in first phase. We did this for nodes in four modules of OR1200[6]. The results are shown in Table 5.3. The first column shows the name of the module. The number of paths that were found to be functionally testable for each module while performing this experiment is shown in the second column. The similar number for functionally redundant paths is given in the third column. The next (fourth) column gives the number of paths that were rejected because the feedback technique found that those paths contained a sub-path that was already deemed non-functional. In the

[5]Within the given bound

[6]Even though a node may belong to a module, the mapping procedure considers the entire OR1200 while checking the path.

last column, we give the node coverage efficiency N. N denotes the percentage of nodes in the module for which the mapping attempt either produced a instruction sequence to test a long path through the node or all the paths through it (given by DATPG) were found to be functionally untestable the given bound. N provides a way of separating the efficiency of the mapping technique from that of the DATPG. N is the fault coverage provided by our mapping technique for the IUFM, if it is given all the paths that it needs, in other words, N would be the actual fault coverage if the DATPG procedure never aborts[7].

Table 5.3: Results for Phase 2 of experiments

Module	Functionally Testable	Functionally Redundant	Rejected Sub-paths	N (%)
or1200_ctrl	1826	29191	68087	90.6
or1200_alu	1427	16985	2716	100
or1200_lsu	970	4077	3744	100
or1200_wbmux	1146	2285	2118	100

We can see from the last column of the table that we achieved 100% coverage efficiency in all the modules except for the `or1200_ctrl` module. `or1200_ctrl` contains the complex decoding logic and the pipeline glue logic. Therefore, the procedure procedure backtracked a lot more times. Even then, we were able to get a coverage efficiency of 90% on this module. The overall coverage efficiency for the four modules is 96%.

[7]We had set a backtrack limit of 10000 for the DATPG procedure

Overall results of the delay test mapping experiments are shown in Table 5.4. It provides the final coverage efficiency and the average time taken for mapping. It also gives the number of times the DATPG procedure hit the backtrack limit of 10000 and the number of times the mapping procedure timed out as a percentage of number of paths considered.

Table 5.4: Overall results

Overview of results	
Final N	96%
Mapping time	18.85s
Backtrack limit reached	2.6%
Timedout	0.7%

5.4 Discussions

In this chapter, we adapted our technique for generating instruction sequences to generate pseudo-robust tests for delay faults in a processor. We introduced an ATPG-based delay test generator, and a feedback mechanism into our instruction mapping technique for this purpose. We demonstrated our technique on the OR1200 processor, where we achieved a coverage efficiency of 96% for delay faults for the four modules that we tested. As pointed out in the previous chapters, the significance of this coverage is increased due to the findings of Maxwell et al. [55].

We mapped the paths into instructions in this chapter. The faulty value has already been propagated to the endpoint of the paths before we

start mapping. This makes it more probable that in many cases, test generated for exciting the path may be sufficient to propagate it to an observable output. Performing excitation only instruction mapping decreases the burden on the mapping considerably (as evidenced by the experimental results shown in Chapter 4). We undertake this experiment and suggest further techniques for efficient test generation in the next chapter.

Chapter 6

Efficient generation of instruction sequences

Generating compact tests is an important factor to reduce the test costs due to the following reasons.

- Test application time (TAT) determines a large part of test cost. As mentioned earlier, loading the cache with data from the tester takes place at a slower speed than the actual running of tests on the processor. Typically, loading of the cache uses the majority of the test application time (TAT) [3]. Reducing the number of instructions is the only way to address this issue.

- Test data volume (TDV) determines the memory requirement of the ATE. ATE costs are highly related to the memory requirements. Therefore, it is imperative to reduce the TDV.

The number of faults in a good delay fault model is typically quite larger than the number of stuck-at faults. Therefore, it is even more important to have efficient test generation in case of delay faults. This problem of efficient test generation for delay faults is handled in this chapter.

In the following sections, we will describe a technique for test generation which makes it more probable that each test detects multiple paths. We also do further experiments upon the approach of the Chapter 5. We used Boolean difference for fault propagation in that Chapter. This might lead to exponential increase in the effort spent on mapping in some cases. We eschew the Boolean difference based approach that ensures propagation and use a path based fault simulation technique for the propagation check. In effect, we use excitation (or controllability) only mapping and check whether propagation of the faulty value has been collaterally ensured by such tests. We then supplement the test generation process by adding a technique that makes each test more probable to detect additional delay faults. This helps in reducing the test content. This methodology of test content reduction is complementary to any code compression based techniques [3].

6.1 Test generation technique

As in Chapter 5, we use IUFM as our fault model and use DATPG for our local true path generation. In the interest of more efficiency, we make a minor modification. We use a threshold of 90% in the first phase and generate a sorted (in the decreasing order of delay) list of longer paths through a node in the second phase.

6.1.1 Instruction mapping procedure

Paths generated by the DATPG procedure are mapped to instructions. In the first phase, all paths generated by DATPG undergo the mapping. In the second phase, the paths are mapped in a sorted order starting from the longest path and stopping as soon as a mappable path is found. The mapping procedure uses the bounded model checker for only controllability and uses simulation for observability. The controllability only mapping also specifies a property of the form $(C \implies O)$. The antecedent C, specifies the same constraints as in Chapter 5. However, the consequent O is trivially false. We do not modify the design.

6.1.1.1 Propagation through simulation

Traditional fault simulation techniques cannot be used in case of delay faults. Fault simulation using *path saboteurs* was proposed in [6]. We adapt that for our technique. Creating *path saboteurs* involves modifying the netlist in such a way that if the path under test is excited, the flop at the end of the path latches in the wrong value. The path saboteur can also made in such a way that it flop latches in the wrong value only when pseudo-robustness conditions are met. An example of a path saboteur (checking for pseudo-robust conditions) is shown in Figure 6.1. The shaded gates are the gates introduced for path saboteur. The original circuit has two AND gates and an OR gate. The path under test has a rising transition passing through these gates. The path saboteur makes sure that if the transition is present at the

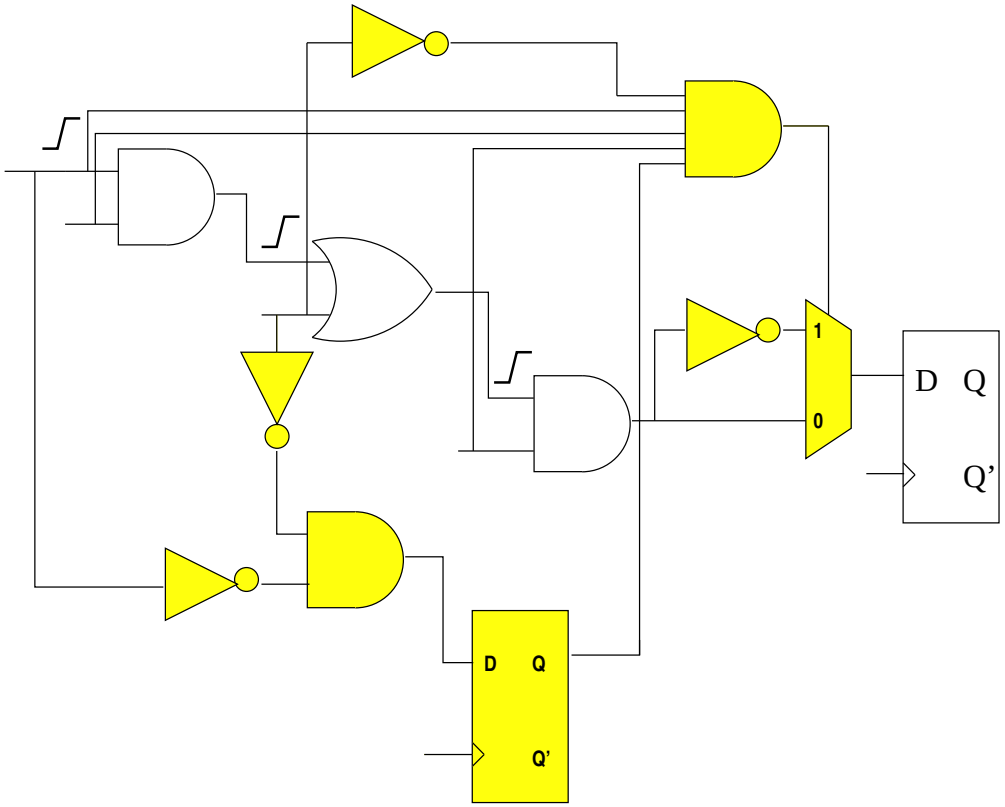


Figure 6.1: Path saboteur checking for pseudo-robust conditions

first gate and pseudo-robustly propagated[1] through the rest of the gates, then the select signal of the multiplexer placed at the end of the path is one. In this case, the flop latches on to the wrong value, and hence the transition will be absent at the output of the path. We take the test sequence generated in Section 6.1.1 and generate a testbench. We simulate this testbench twice, once when the netlist has not been sabotaged and once with the sabotage. If the outputs of the circuit are different in these two simulations then the path has

[1]Note that OR gate's side-input is checked for values in both cycles

been detected.

6.2 Test content reduction

Using IUFM reduces the number of faults to be detected when compared to using the path delay fault model. However, the number of faults targeted can still be large. Hence, test content reduction is important. We attempt to reduce the test content in two different ways: a) fault simulation b) increasing probability of multiple path detection during test generation. While attempting test content reduction, we target only those paths for which a test was produced by using the technique outlined in Section 6.1. We call these paths as critical paths.

6.2.1 Test content reduction via simulation

In this part of the technique we identify all the critical paths that can be detected by the tests that were generated, using fault simulation. If we use the simulation technique in Section 6.1.1.1 as is, we would need one good netlist simulation and one sabotaged netlist simulation per critical path for each test. Therefore, if there are n critical paths we need $n + 1$ simulations per test. To reduce the need for n sabotaged netlist simulations we propose a new kind of path saboteurs. We call these the *excitation saboteurs* because they check only for path excitation. An example is shown in Figure 6.2. The difference between the original path saboteurs (Figure 6.1) and the excitation saboteurs (Figure 6.2) is the absence of the multiplexer in the excitation

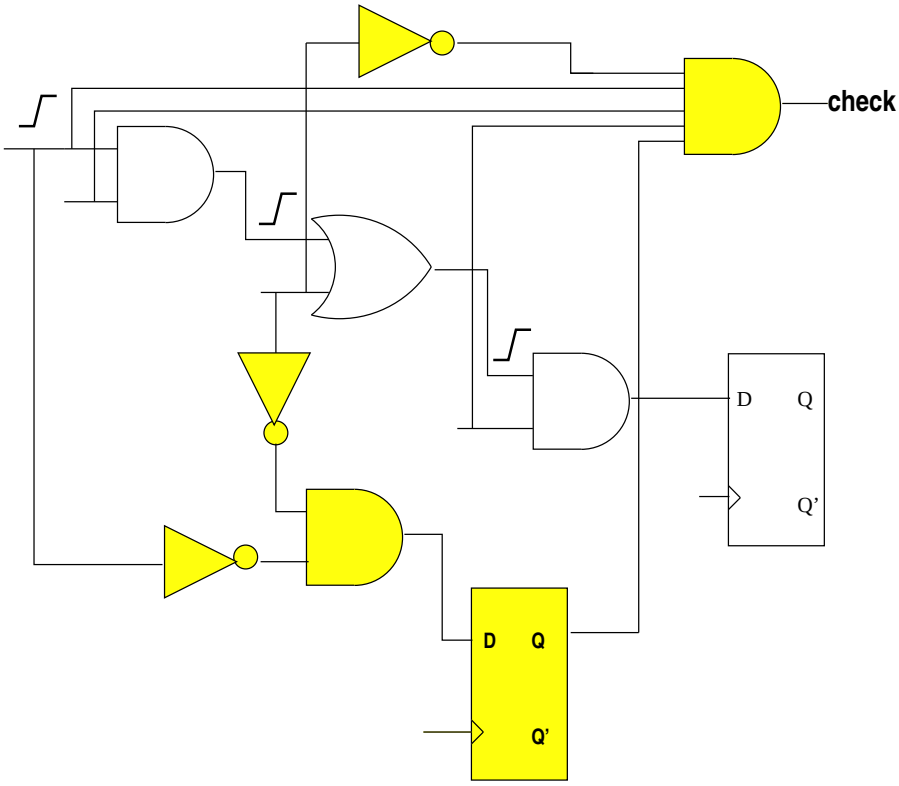


Figure 6.2: Path saboteur checking for excitation only pseudo-robust conditions

saboteur. This makes sure that the faulty value is not latched on to the flop. The *check* signal goes high when the path is excited with pseudo-robust conditions. The advantage of the excitation saboteur is that many paths can be checked for excitation together in one simulation by monitoring the check signals corresponding to each path. For every test generated, we create excitation saboteurs for all the paths and identify the paths that are excited by the test. We then create full saboteurs for those paths individually and verify whether those paths are also propagated by the test. The algorithm for this is given below.

```
STEP 1: Choose a test and create a testbench
STEP 2: Create excitation saboteurs for all paths
STEP 3: Simulate the sabotaged circuit
STEP 4: Store response R. Identify the paths excited
STEP 5: Repeat for each excited path P steps 6 through to 8
STEP 6: Create full path saboteur for P
STEP 7: Simulate circuit
STEP 8: Compare response to R. If responses different path P is
        detected
```

6.2.2 Multiple path targeting

Now that we have found all the critical paths detected by the tests, we try to increase the probability that each test detects more paths. We do this by changing the constraints given to the bounded model checker. Note that the

pseudo-robust conditions allow the side-inputs to transition if the on-input transitions to a non-controlling value. We take advantage of this by making these side-input transitions a requirement in the constraints given to the bounded model checker. Therefore, the test generated by the mapping technique will have more transitions, increasing the probability that some other critical path will now be pseudo-robustly detected. However, if we specify this on all feasible side-inputs, it becomes highly likely that such constraints cannot be satisfied and test cannot be generated. Therefore, we try to identify the side-inputs that can be allowed to transition with minimal effects on the testability of the path. We do this by gathering the support-set of each side-input. The support set of a signal is all the signals that are in the backward combinational cone of influence of the signal. We allow a side-input to transition iff its support set is independent of the support sets of all other side-inputs and the nodes on the path. Computation of the support sets is fairly straight forward using a depth first search algorithm. Therefore, the computation overhead is very low.

6.3 Experimental results

We performed our experiments on the OR1200 processor. We used the same synthesized design as in Section 5.3. We also used the same bound (7) as in Section 5.3. We used a threshold of 90% of the clock cycle for the first phase of testing. The results of first phase are given in Table 6.1. The first column shows the number of paths that were found to be above the threshold

Table 6.1: Results of Phase 1 of experiment

No. of Paths	Functionally Testable	Functionally Untestable	Excitation Only	P (%)
2998	1891	1105	2	99.9

by DATPG, which were 2998 in number. As shown in the second column 1891 of these paths was found to have a test, whereas 1105 of the paths did not have test within the given bound (shown in the third column). The fourth column shows that, for 2 paths, our technique found a test for excitation. However, these two tests did not guarantee propagation. The fifth column gives the path efficiency P. We calculate the path efficiency as follows.

$$P = 1 - \frac{E}{T} * 100 \tag{6.1}$$

where E denotes the number of paths for which we found excitation only sequences and T is the number of paths for which mapping was attempted. Therefore, P gives the frequency with which the new technique gives a result, *i.e.*, either gives a test which detects (excites and propagates) the path or computes the path to be functionally untestable within the given bound. This is nearly 100% as shown in the last column.

All the nodes in the paths for which a test generated in the first phase were denoted as covered. In the second phase, we generated the longest functionally testable path through each node not covered in first phase. We did this for nodes in four modules of OR1200. The results are shown in Table 6.2. The first column shows the name of the module. The second column gives the number of paths in each module that were found to be functional. Note

that we stop the process for a node when a functional path is found through it going in decreasing order of path delays. Therefore, this column also shows the nodes covered in the second phase. The third column gives the number of paths for which the tests we found only excited the path. The fourth column gives the number of paths on which the instruction mapping was attempted per each module. The fifth column gives the path efficiency P. In the last column, we give the node coverage efficiency N, calculated as explained in Section 5.3. The overall number are 96% and 98% for P and N respectively.

Table 6.2: Results for Phase 2 of experiments

Module	Functional Paths	Excitation Only	Total Paths	P (%)	N (%)
or1200_ctrl	619	4554	110242	96	93
or1200_alu	365	510	2181	77	99
or1200_lsu	124	0	3138	100	100
or1200_wbmux	364	267	27345	99	99

Important result to be noted here is the fact that the mapping always produced a result, *i.e.*, there were no timeouts. One other result not shown in the table is we got a lower bound on transition fault coverage, which was 92% for the four modules. We derived this by parsing through the paths produced by the mapping technique and identifying the nodes in all the paths. This is a lower bound on the transition fault coverage because the test for a path might have toggled other nodes (which are not part of critical paths), that were not monitored. Note that this number is less than the node coverage efficiency N because of the aborts in the DATPG due to the backtrack limit that we

set (which was 1000). The stuck-at fault coverage is also at least 92% since transition faults are a superset of the stuck-at faults.

We applied both the test content reduction techniques that were described in Section 6.2. We started with 3388 tests that were generated during both phases of the test generation. The first technique was to use the iterative path saboteur based fault simulation technique described in Section 6.2.1. We found that this reduced the number of tests to 1688. We generated tests for the paths using the technique described in Section 6.2.2. We found that there are only limited number of critical paths which have side-inputs with no common shared backward logic with any other side-input. Therefore, we augmented the technique with more iterations. To do so, we sorted the side-inputs in decreasing order of number of side-inputs each one shared logic with. After each iteration, we dropped the top node in this sorted list and found the nodes which now do not share any logic with any other side-input in the list. We did this three times. Iteration 0 was our base technique with no augmentation. The results of this are given in the Table 6.3. The first column in the table gives the iteration number. The second column gives the number of paths that have some side-input which can be forced to have a transition at each iteration. The third column gives the number of paths that were found to be functionally excitable. The fourth column gives the number of paths that were also propagated to an observable point. We used these tests and again applied the technique described in Section 6.2.1. We found that this reduced the number of tests required to detect the critical paths to 1676. The further

Table 6.3: Results of each iteration of multi-path targeting

Iteration	No. of Paths	Excited	Functional
0	1439	657	562
1	2388	1490	1004
2	2823	1906	1507
3	3462	2352	1939

compression achieved is low because the original tests that were generated were good at detecting multiple paths. The overall test content reduction that was achieved was **49.4%**.

The test content reductions described are complementary to code compression techniques. To demonstrate this we applied some code compression techniques on the 1676 tests that were found in the previous experiment. The following encoding schemes were used: Huffman encoding, run length encoding and lzw encoding. An 8-bit symbol length is used for the Huffman encoding. Both character level and bit level encoding schemes were implemented for run length encoding. Runs of bytes were encoded in character level run length encoding. In binary run length encoding, 2-bit and 3-bit codewords encoding runs of 0's were used. 2-bit and 3-bit codewords encoding runs of 0's and runs of 1's alternatingly were also used for bit level encoding. For lzw, window sizes of 256 bytes, 1024 bytes, 4096 bytes and 16384 bytes were used. To estimate the theoretical limit to the compressibility of the instruction sequences, we also calculated the entropy of the sequences using a symbol size of 32 bits. The results are shown in the graph of Figure 6.3. Compressions shown in the graph also include the compression provided by our technique. We see that

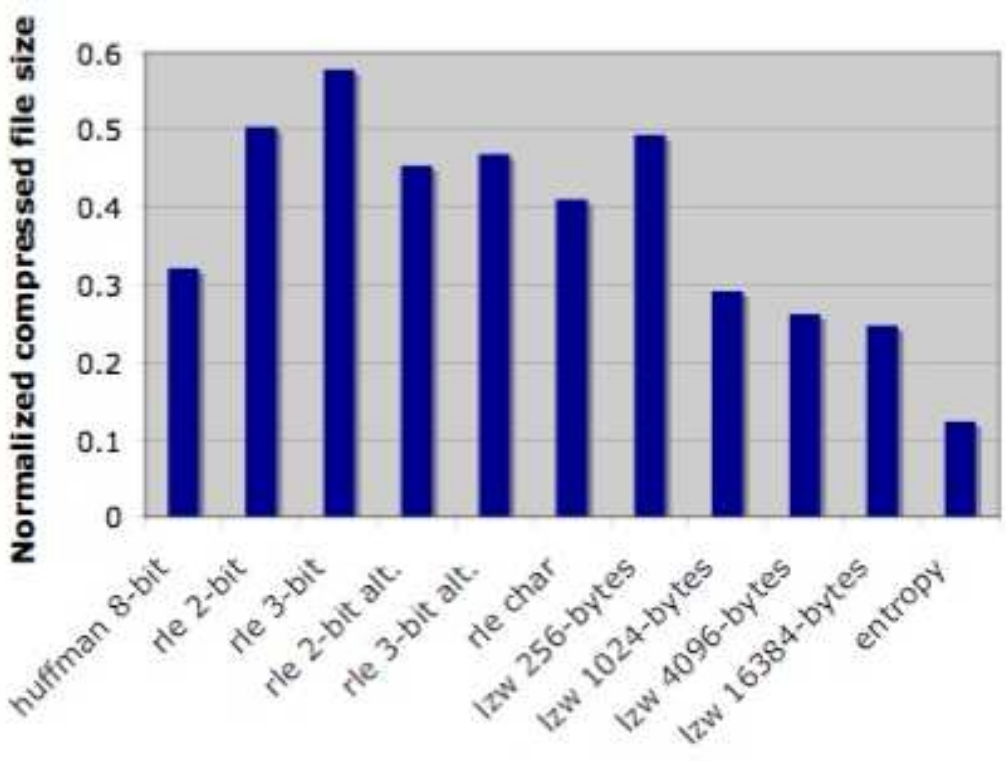


Figure 6.3: Compression achieved by code compression techniques

we compress the tests to 25% of original size. Further information regarding many of these code compression technique and the overheads involved can be obtained in [3].

6.4 Discussion

In this paper, we developed a technique for efficiently generating compact instruction sequences to pseudo-robustly test a processor for delay faults. We demonstrated our technique on an OR1200 processor, where we achieved a coverage efficiency of 98% of delay faults on four modules. The test content

reduction techniques compressed the tests from 3388 to 1676, a reduction of 50.6%. The advantage of these test content reduction techniques is that they do not involve any memory/performance overhead during the test application process. Moreover, these techniques are complementary to any existing code compression technique, which was also demonstrated (achieving an overall compression of 75%).

We found that majority of the tests generated detected multiple paths without any additional constraints. One of the reasons for this is the netlist was not optimally synthesized. Therefore, most of the critical paths go through same part of the design. We expect that our multiple path targeting technique will achieve a lot of further content reduction on more optimized designs.

Chapter 7

Testing the non-processor cores of an SOC

In this chapter, we will present a comprehensive solution to test an embedded processor based SOC. The techniques outlined in the previous chapters for testing a processor can be used to test the embedded processor of an SOC. Therefore, we will now direct our efforts towards providing a solution to test the other cores in an SOC.

We will use the processor to deliver the tests and collect the responses, in other words, the processor will be our test source and sink. Our technique does not need any boundary scan [78] or any wrappers [45] around the cores being tested. This gives us the advantage of not requiring any modifications to the SOC design. Moreover, with the processor accessing the cores in the normal operating manner, the test will be at-speed.

As described in Chapter 2, others have proposed techniques for testing the SOC using processor with functional access. However, unlike those methods our technique very suited for black-box testing of the other cores. Our technique does not need any knowledge of the internal logic of a core. It needs only the sequences that can test the IP core. Since most of the other cores are bought from intellectual property vendors, they are generally available only as

black-boxes. Therefore, black-box testing of IP cores is important.

The test vectors that we need for our technique might be of any kind. It might be the validation vectors used during the design of the cores or any other vectors that can test the IP core functionally. We parse those vectors and map them into instructions which when executed by the embedded processor will produce those vectors at the inputs of the non-processor cores.

The technique detailed in the following sections can also be used to test IP cores that are available as white-box or grey-box. In those cases, we can even have a hierarchical approach. In the hierarchical approach we can generate the test vectors using ATPG tools and then map them by the proposed method.

We will illustrate the effectiveness of the technique using an SOC that we designed. The SOC has an ARM core, memory and an AES cryptography core. We map some testbenches of the AES core and show that the coverage that we achieve in the SOC environment is within 0.5% of coverage achieved by the testbenches on just the core.

7.1 Our SOC testing methodology

Block diagram of an SOC is shown in Figure 7.1. As shown in the figure, in general an SOC has a embedded processor communicating to other cores via a system bus. In general, the other cores are the intellectual property of various vendors chosen according to the intended application of the SOC. Each IP

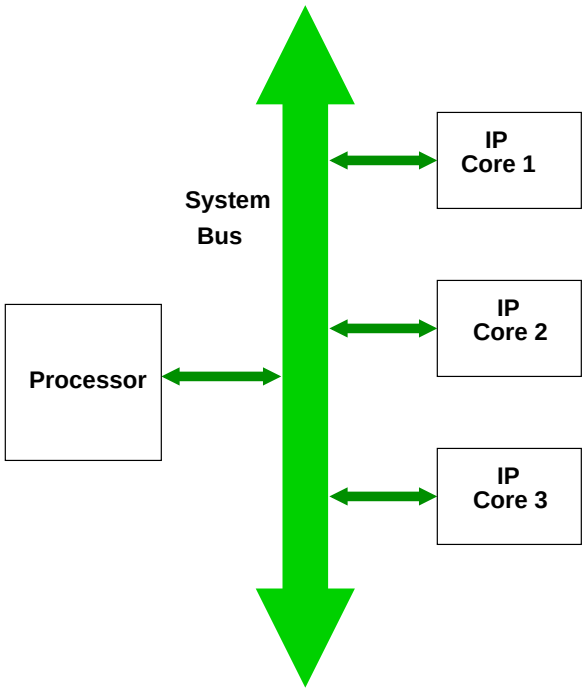


Figure 7.1: Design of a generic SOC

core has a corresponding driver program written in a high level programming language like C which allows the processor to communicate with it. Those driver programs take in data values to be sent to the IP core as input during the transmit operation and then produce the set of instructions that make the processor deliver those values to the IP core through the bus. During the receive operation the driver program will read in the data at the output of the IP core and make it available to the processor via the bus.

Our technique for SOC testing is shown in the form of a block diagram

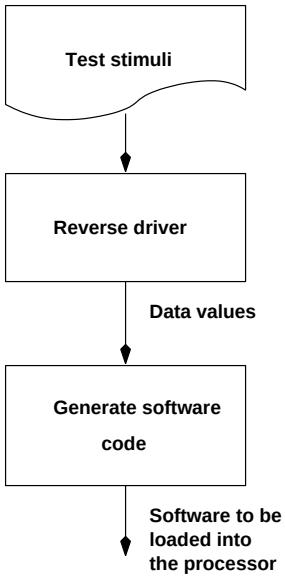


Figure 7.2: SOC test flow proposed by our technique

in Figure 7.2. We start with a set of test stimuli. These test stimuli can be gathered from anywhere. It may be the test vectors provided by the IP vendor in case of a black-box core or it may be verification/validation vectors generated during the design of the core. In case no such vectors exist, it might be generated by applying ATPG algorithms to the core. Irrespective of the source of the vectors, we use them as the starting point in our SOC test flow. We use these vectors as the input to our reverse driver program.

7.1.1 Reverse driver

A driver program takes in data to be sent to the core and produces the instructions that will send the data to the core. Our reverse driver takes in the

Table 7.1: Description of the registers inside the core

Address	Functionality
0	Control register
1	Data bits 31 to 24
2	Data bits 23 to 16
3	Data bits 15 to 8
4	Data bits 7 to 0

test stimuli as it is supposed to appear at the inputs of the core and provides the data values that can be give to the driver program to get the instructions that will send the test stimuli to the core. The reverse driver program is specific to a core like the driver program. For the purposes of illustration, let us suppose that the IP core being tested is a peripheral device that communicates with the external environment. Let us suppose it can send/receive 32 bits of data at 4 different baud rates. Assume that the bus interface allows the communication to be of only 8-bits address/data between the IP core and the processor. To facilitate communication with the processor (via the bus) the IP core has an arrangement of five registers that have functions as described in Table 7.1. The control register at address zero specifies the speed (last 2 bits) of transfer and also whether the operation is send/receive (third least significant bit). The rest of the registers provide the data that is sent/received. Table 7.2 shows an example test sequence provided by the vendor. Our reverse driver for this core will parse this test sequence and identify that it requires a send operation at the third speed rate from the data sent to the address 0x00. From the rest of the test sequence it will get the data to be sent, which is 0x54DF7178. This

Table 7.2: A test sequence provided by the vendor

Address	data
0x00	0x07
0x01	0x54
0x02	0xDF
0x03	0x71
0x04	0x78

data will then be given to the next step of our technique.

7.1.2 Generating the software to be loaded into the processor

The reverse driver program has parsed through the test sequence to decipher the data to be sent to the core. Now we have to write the software that does that. We know that the driver program for a core takes in the data that needs to be sent to the core and provides the instructions to do so. We use this functionality of the driver program for our purpose. We give the data generated by the reverse driver to the driver program of the core. The driver program will then generate the instructions necessary to produce the test sequence at the core inputs.

The driver program is in general provided for each core or is written by the designers as part of the SOC development process. Therefore, the writing of the driver program is not an overhead associated with our methodology. Our reverse driver program will have to parse through the test stimuli and produce data values in a format readable by the driver program. This is the only overhead associated with our process.

7.2 Simulation and coverage measurement

Now that we have a methodology to generate the software to be loaded into the processor for test, we need a technique to measure the coverage of the resulting sequences. For this purpose, we use the generally used simulation and fault simulation techniques. We first simulate the platform after loading the instructions we have generated into the processor. During this simulation we monitor the inputs of the core and grab the signals that arrive there. Once the simulation is done, we fault simulate the core alone using the vectors that we have grabbed at the boundaries.

This technique for fault coverage estimation is generic and portable to any SOC. The simulation that we use to grab the vectors at the input of the cores is the popular methodology used while validating the SOC design. We also use the normally used fault simulation methods to arrive at our coverage. We use the fault simulation on the core alone. This makes fault simulation feasible on large SOC designs. Note that gate level structural fault simulation is possible only for grey-box or white-box cores. For black-box cores, we can attempt a functional fault simulation.

7.3 Implementation

To show the effectiveness of our methodology we implemented an SOC using an ARM core [35], memory and an AES core available from opencores [89]. The block diagram of the implemented SOC is shown in Figure 7.3. As shown in the figure, the cores in the SOC are connected together by the

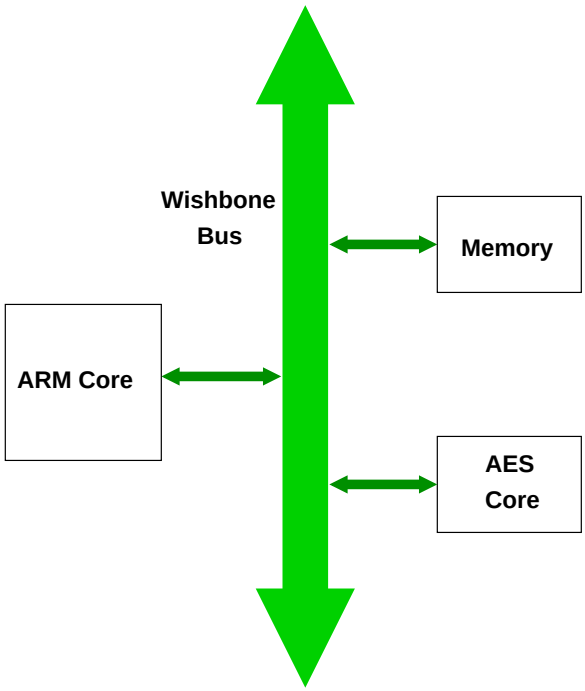


Figure 7.3: SOC containing ARM and AES cores

Wishbone interface (also available through opencores). We use the 128-bit AES core which takes in 128-bit data and key for encryption and decryption. The AES core was memory mapped while integrating with the SOC. The Wishbone interface allows 32-bit transmission through it. We will now illustrate our technique on this SOC.

7.3.1 Reverse driver for the AES core

The AES interface with Wishbone facilitates the transfer between 128-bit AES core and the 32-bit Wishbone interface by having 32-bit registers

reset	aes_wb_stb_i	aes_wb_we_i	aes_wb_cyc_i	aes_wb_adr_i	aes_wb_dat_i
1	1	1	1	0x0004	0x3462
1	0	0	0	0x0004	0x3462
1	1	1	1	0x0008	0xCD23
1	0	0	0	0x0008	0xCD23
1	1	1	1	0x000C	0xAB12
1	0	0	0	0x000C	0xAB12
1	1	1	1	0x0010	0x1289
1	0	0	0	0x0010	0x1289
.	.	.	.	.....	.....

Figure 7.4: A test sequence

to store the data. There are a total of 9 registers, 4 each to hold the data and the key values, and one to act as the control register. Separate addresses are assigned to each of these registers. An example sequence that occurs during the simulation of our SOC is given in Figure 7.4. `aes_wb_stb_i`, `aes_wb_we_i`,`aes_wb_cyc_i` are all control signals from the Wishbone bus indicating whether the core has been activated and whether the current operation is read/write. The `aes_wb_adr_i, aes_wb_dat_i` are the address and the data signals respectively. Our reverse driver will parse this sequence and whenever the core is activated will parse the address and data signals to correctly identify what data is to be sent in. Parsing the the example sequence the reverse driver will interpret the data of 0x3462CD23AB121289 to be sent to the AES.

7.3.2 Generating the software to be sent to AES

While designing the SOC, we also designed driver programs that take in data or key to be sent to the core and generates an instruction sequence

Table 7.3: Information regarding the AES core implemented

No. of inputs	69
No. of outputs	33
No. of combinational primitives	9225
No. of sequential primitives	1119
No. of uncollapsed faults	64070

using the ARM ISA to send those values to the AES core. During our test generation process, we give this driver program the outputs generated by the reverse driver program. The validation sequence might also necessitate the reading in the values generated by the AES core. We also have driver programs for that purpose which we use.

7.4 Experiments

As described in Section 7.3, we implemented an SOC using an ARM core and an AES cryptography core. The AES core and ARM processor were both written in Verilog. The AES core also had wrapper allowing it to interact with the Wishbone interface. We synthesized the AES core using a 0.18u technology. The data regarding the synthesized AES core is given in Table 7.3.

To validate our SOC design we wrote a few testbenches. These testbenches generated random 128-bit values for data and key. We used the same key to encrypt and decrypt the result of encryption or vice-versa and checked

Table 7.4: Results of the SOC testing experiment

	Size (bytes)	Fault coverage(%)	Original Coverage(%)	No. of Cycles	Original no. of Cycles
Testbench 1	7808	90.01	90.26	6700	6373
Testbench 2	9128	90.15	90.35	7816	7435
Testbench 3	10432	90.2	90.44	8932	8496

whether we ended up with original 128-bit data. We generated many such sequences. We used these testbenches as the sequences that we need to map for SOC testing. We mapped these test sequences into ARM instructions as described in Sections 7.3.1 and 7.3.2. We then simulated the entire ARM based SOC using the instructions generated and captured the signals at the input of the core. We fault simulated the AES core using the captured signals using a commercially available fault simulation tool [20]. The results of the experiment are shown in Table 7.4.

The first column in Table 7.4 gives the testbench number. We used the three of the testbenches that we generated to validate our testing methodology. The second column in the table gives the size of the code generated when these testbenches were mapped to instructions using our technique. The size is given in bytes. This size does not show the amount of code that may be necessary to dump the register file and other values (during actual test application) so that they might be compared against the golden values derived during the simulation. The third column gives the amount of fault coverage achieved by each code when simulated and fault simulated according to our methodology.

The numbers in the fourth column give fault coverage of the original sequences when applied directly to the AES core. We can see that the fault coverage that we obtain is very close to the original fault coverage numbers. The fifth and sixth columns give the number of cycles in the sequences generated by our methodology and originally in validation sequences respectively. We can see that our methodology increases the number of cycles by around 400 cycles on an average because of the transactions involved in sending the data through the Wishbone bus.

7.5 Discussion

We presented our technique for testing SOCs. Even though we have explained our technique for an SOC, it can also be used to test a network-on-chip (NOC). Our technique involved mapping the test/validation sequences that may be provided by the vendor or generated by the user into software code to be loaded onto the embedded processor. The advantage of using the embedded processor as the test source and sink is that it avoids the need for any modifications in the design. The code can be loaded onto the processor by the mechanisms like JTAG or ethernet using which the SOC normally communicates with the external environment. The methodology is highly suitable to black-box testing of IP cores since we do not need any knowledge of the internal logic of the IP core being tested. This is a very important contribution considering the increasing trend towards design reuse.

The technique can also be used for grey-box or white-box cores. In those

cases a very high coverage can be attained by using a hierarchical approach. We can generate tests for the cores using any ATPG tool and then map them into instructions using our technique. We showed the effectiveness of our technique by implementing an SOC using ARM and AES cores. We had an insignificant loss of coverage when the testbenches were mapped to software code.

The coverage obtained by our technique highly depends on the sequences that we start from. This is both a pro and a con of our technique. On one hand, this makes our technique suitable for black-box testing. On the other hand, if the sequences are not good then the coverage obtained can be low. One way of overcoming it is by using a hierarchical approach for even black-box cores. We can derive a functional representation of the core based on the description given and generate sequences that provide good coverage of this representation (like the technique for testing processors described in [77]). These sequences can then be mapped using our technique.

Chapter 8

Conclusion

In the preceding chapters, we explained our techniques for generation of at-speed functional tests targeting structural faults in processors and SOCs. As detailed in Chapter 1, we have addressed an important problem that has become even more pressing with the emerging process technologies. We initially applied our technique to target stuck-at faults and proceeded to delay faults later. We also presented a comprehensive solution to test a SOC. We achieved better than 90% coverage of faults. This is very useful given the fact that at-speed functional tests with lower coverage are more effective than the slower structural tests with higher coverage [55].

The technique that we have proposed for test generation targeting faults in a processor is based on a hierarchical approach. The hierarchy gives this approach quite a few advantages. It allows us to use traditional structure based test generation techniques at the local level where the structure needs to be used to include the fault information. It also gives us the freedom to utilize a verification engine which is more suited to take in the instruction set based constraints at the global level. We also introduced a feedback between these two levels to attain higher coverage.

We also introduced a methodology for testing non-processor cores in SOCs. We used the intelligence of the embedded processors in SOCs for this purpose. The technique that we explained is highly suited to test black-box cores that are part and parcel of the design reuse paradigm. We use any vectors that are available to test the core and map them into instructions that can be loaded into the embedded processor. We will now discuss some of issues with our technique and any future directions in related areas that seem promising.

8.1 Testing for stuck-at faults

We had defined the modules in the design as the boundaries for the local level test generation in case of stuck-at faults. This facilitates using tools that operate at the RT-Level for global mapping because the signals at module boundaries, *i.e.*, the module inputs and outputs, are visible both at the RT-Level and the gate level. Our properties for test generation are also specified at the RT-Level. We do not develop a tool from scratch, rather we use existing tools to our advantage. In this scenario, once a RT-Level constraint solving tool is developed and shown to work it can be harnessed for our purpose. In fact, any tool that can take in constraints and solve them on a model can be used in our methodology. This leaves us open to use any word level reasoning tool or any other such developments.

One of the drawbacks of using a hierarchical approach is the fact that local test generation does not have the knowledge of constraints while attempting the test generation. We overcome this by having a feedback that

we introduced for delay faults. The coverage we achieved for delay faults is the lower bound on the coverage that we can obtain with the same tests for stuck-at faults. However, even with feedback, the test generation's effectiveness is reduced when the local test generator keeps producing tests that are not globally mappable. One way of overcoming this would be to extract some of the constraints *a priori* and pass them to the local test generation tool. Techniques like symbolic simulation [13] can be used for this purpose. The model can be symbolically simulated using the instruction set and constraints can be extracted from the trace produced. Another technique that can be used is instruction slicing [85] that slices the RT-Level code based on the way each instruction affects the code.

8.2 Delay faults

We have proposed a very effective technique for testing of delay defects in a processor. We used a fault model called the improved unified fault model. This model allows us to target the small delay defects without having to test for all the paths in the design. However, the fault model depends on the timing information provided by the synthesis tools. With the new process technologies the delays in the silicon differs highly from what was projected by the pre-silicon tools. Therefore, a fault model which takes such vagaries into account will be a very important contribution.

A fault model based on switching activity will be an interesting approach. A path which is long according to the pre-silicon tools may not be

that long in the design. However, any switching activity that is introduced by a vector in pre-silicon tool will also be present in the actual silicon. The effects such as crosstalk, delay variation due to the heat in circuit depend on the switching activity in the regions of the circuit. Therefore, if we generate vectors that increase the switching in various regions we might catch the paths that affect the timing of silicon much better.

While generating tests for delay defects we used the pseudo-robustness conditions which mimic the robustness conditions except for allowing glitches on the side-inputs. This allows us to target more paths than what would have been possible if robustness conditions had been used. We can further relax this conditions in certain cases to make it non-robust so that we can get better coverage and also get better screening tests for silicon.

8.3 Scalability and portability

One of the main questions regarding the technique is its scalability. The scalability of the technique depends on the effectiveness of the tool being used for global mapping. Formal verification engines have the problem of state space explosion. It is overcome in the case of bounded model checkers (the kind of tool that we use) by unrolling only up to a specific number of cycles. Moreover, the kind of properties that we use to ascertain controllability is relatively simple compared to the properties being used in the design verification phase, mainly because the temporal implication that our properties have is very limited. Moreover, the signals being checked in the property have a

structural relationship since they were assigned values to test for a single fault. This makes our technique applicable to larger designs. We can also improve the scalability by using techniques like abstraction or slicing. Another way for improving the scalability would be to use multiple levels of hierarchy. We can have separate engines for test generation at the module level, cluster level chip level that communicate with each other. These engines can also take in different representations of the design. For example, the module level testing can be done at gate level, the cluster level testing using RT-Level design and the full chip testing using a C like representation.

The methodology that we have proposed is portable to any design. We have used tools that are part of the existing design flow. The knowledge that is required of the user is not intensive. There will be a initial setup required when the constraints imposed by the instruction set and other related signals are extracted. This can be done parsing through the architecture document. This can be also be automated, which is an exciting area of future research.

8.4 SOC testing

SOC testing is a very fertile area of research which has been increasing in importance in the past few years. We have proposed an approach highly suited to test the black-box cores and can also be easily used for grey-box or white-box cores with a hierarchical approach. In case of black-box cores, the dependance on the existence of test sequences can also be avoided by an hierarchical approach that uses test generation on a functional representation

approach of [77] to generate the sequences.

The technique as proposed is directly applicable to test for stuck-at faults. To target delay faults some modifications may be necessary. This is because the clock's absence when the core is not active will not affect stuck-at faults but it might affect delay faults. This can be overcome by having vectors test for delay faults only when the core is already active through previous sequences. The proposed technique does not need to know the internals of the core being tested, hence it can be used to test SOCs containing analog, MEMS or any other non-digital cores that are becoming common occurrences with the increasing trend towards heterogenous SOCs.

Bibliography

[1] Semiconductor Industry Association. The international technology roadmap for semiconductors, 2005 edition. http://www.itrs.net/, 2005.

[2] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *International conference on Computer-aided design*, pages 188–191, 1993.

[3] K. J. Balakrishnan, N. A. Touba, and S. Patil. Compressing functional tests for microprocessors. In *Asian Test Symposium*, pages 428–433, 2005.

[4] P. H. Bardell, W. H. McKenney, and J. Savir. *Built-in Test for VLSI: Pseudorandom Techniques*. John Wiley and Sons, New York, 1987.

[5] R. Bencivenga, T. J. Chakraborty, and S.Davidson. The architecture of the GenTest sequential test generator. In *Custom Integrated Circuits Conference*, pages 17.1.1–17.1.4, 1991.

[6] P. Bernardi, M. Grosso, and M. S. Reorda. Hardware-accelerated path-delay fault grading of functional test programs for processor-based systems. In *Great lakes symposium on VLSI*, pages 411–416, 2007.

[7] S. Bhatia and N. K. Jha. Integration of hierarchical test generation with behavioral synthesis of controller and data path circuits. *IEEE Transactions on VLSI Systems*, 6(4):608–619, 1998.

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[9] L. Bolzani, E. Sanchez, and M. S. Reorda. A software-based methodology for the generation of peripheral test sets based on high-level descriptions. In *Conference on Integrated circuits and systems design*, pages 348–353, 2007.

[10] L. Bolzani, E. Sanchez, M. Schillaci, M. S. Reorda, and G. Squillero. An automated methodology for cogeneration of test blocks for peripheral cores. In *International On-Line Testing Symposium*, pages 265–270, 2007.

[11] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Design automation conference*, pages 338–342, 2003.

[12] D. Brahme and J. A. Abraham. Functional testing of microprocessors. *IEEE Transactions on Computers*, 33(6):474–485, 1984.

[13] R. E. Bryant. Symbolic simulation - techniques and applications. In *Design automation conference*, pages 517–521, 1990.

[14] A. Carbine. Scan mechanism for monitoring the state of internal signals of a VLSI microprocessor chip. US Patent Number 5,253,255, 1993.

[15] J. L. Carter, V. S. Iyengar, and B. K. Rosen. Efficient Test Coverage Determination for Delay Faults. In *International Test Conference*, pages 418–427, 1987.

[16] T. J. Chakraborty, V. D. Agrawal, and M. L. Bushnell. Path delay fault simulation of sequential circuits. *IEEE Transactions on Very Large Scale Integrated Systems*, 8(2):223–228, 2000.

[17] L. Chen, S. Ravi, A. Raghunathan, and S. Dey. A scalable software-based self-test methodology for programmable processors. In *Design Automation Conference*, pages 548–553, 2003.

[18] W. Cheng and M. Yu. Differential fault simulation for sequential circuits. *Journal of Electronic Testing: Theory and Applications*, 1(1):7–13, 1990.

[19] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero. Fully automatic test program generation for microprocessor cores. In *Conference on Design, Automation and Test in Europe*, pages 1006– 1011, 2003.

[20] Mentor Graphics Corporation. Flextest^{TM}. http://www.mentor.com.

[21] B. I. Dervisoglu and G. E. Stong. Design for testability: Using scanpath techniques for path-delay test and measurement. In *International Test Conference*, pages 365–374, 1991.

[22] E. B. Eichelberger and T. W. Williams. A logic design structure for LSI testability. In *Design Automation Conference*, pages 462–468, 1977.

[23] F. Fallah, P. Ashar, and S. Devadas. Simulation vector generation from HDL descriptions for observability-enhanced statement coverage. In *Design automation conference*, pages 666–671, 1999.

[24] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. *IEEE Transactions on Computers*, 32(12):1137–1144, 1983.

[25] P. Gelsinger. Discontinuities driven by a billion connected machines. *IEEE Design and Test of Computers*, 17(1):7–15, 2000.

[26] I. Ghosh and M. Fujita. Automatic test pattern generation for functional RTL circuits using assignment decision diagrams. In *Design Automation Conference*, pages 43–48, 2000.

[27] I. Ghosh, A. Raghunathan, and N. K. Jha. A design for testability technique for register-transfer level circuits using control/data flow extraction. *IEEE Transactions on Computer-Aided Design*, 17(8):706–723, 1998.

[28] P. Goel. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computers*, 30(3):215–222, 1981.

[29] A. J. V. D. Goor and T. J. W. Verhallen. Functional testing of current microprocessors. In *International Test Conference*, pages 684–695, 1992.

[30] A. J. Van De Goor. Using march tests to test SRAMs. *IEEE Design and Test*, 10(1):8–14, 1993.

[31] S. Gurumurthy, S. Vasudevan, and J. A. Abraham. Automated mapping of precomputed module-level test sequences to processor instructions. In *International Test Conference*, page 12.3, 2005.

[32] S. Gurumurthy, S. Vasudevan, and J. A. Abraham. Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor. In *International Test Conference*, page 27.3, 2006.

[33] S. Gurumurthy, R. Vemu, J. A. Abraham, and D. G. Saab. Automatic generation of instructions to robustly test delay defects in processors. In *European Test Symposium*, pages 173–178, 2007.

[34] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski. Logic BIST for large industrial designs: real issues and case studies. In *International Test Conference*, pages 358–367, 1999.

[35] J. K. Huggins and D. V. Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):563–580, 1998.

[36] S. Hwang and J. A. Abraham. Reuse of addressable system bus for soc testing. In *International ASIC/SOC Conference*, pages 215–219, 2001.

[37] V. Immaneni and S. Raman. Direct access test scheme-design of block and core cells for embedded ASICs. In *International Test Conference*, pages 488–492, 1990.

[38] A. Jas, J. Ghosh-Dastidar, and N. A. Touba. Scan vector compression/decompression using statistical coding. In *VLSI Test Symposium*, pages 114–120, 1999.

[39] R. Jayabharathi. *Hierarchical timing verification and delay fault testing.* PhD dissertation, The University of Texas, 1999.

[40] K. Jayaraman, V. M. Vedula, and J. A. Abraham. Native mode functional self-test generation for systems-on-chip. In *International Symposium on Quality Electronic Design*, pages 280–285, 2002.

[41] T. Kirkland and M. R. Mercer. A topological search algorithm for ATPG. In *Design automation conference*, pages 502–508, 1987.

[42] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Effective software self-test methodology for processor cores. In *Conference on Design, automation and test in Europe*, pages 592–597, 2002.

[43] A. Krishnamachary and J. A. Abraham. Test generation for resistive opens in CMOS. In *Great Lakes symposium on VLSI*, pages 65–70, 2002.

[44] A. Krstic and K. Cheng. *Delay Fault Testing for VLSI Circuits.* Kluwer Academic Publishers Boston, Ma, 1998.

[45] A. Krstic, W. Lai, K. Cheng, L. Chen, and S. Dey. Embedded software-based self-test for programmable core-based designs. *IEEE Design and Test*, 19(4):18–27, 2002.

[46] Cadence Berkeley Laboratories. BMC engine of symbolic model verifier. http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/.

[47] W. Lai, A. Krstic, and K. Cheng. On testing the path delay faults of a microprocessor using its instruction set. In *VLSI Test Symposium*, pages 15–20, 2000.

[48] W. Lai, A. Krstic, and K. Cheng. Test program synthesis for path delay faults in microprocessor cores. In *International Test Conference*, pages 1080–1089, 2000.

[49] T. Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, 1992.

[50] C. J. Lin and S. Reddy. On delay fault testing in logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):694–703, 1987.

[51] Y. Lin, F. Lu, K. Yang, and K. Cheng. Constraint extraction for pseudo-functional scan-based delay testing. In *Asia South Pacific design automation conference*, pages 166–171, 2005.

[52] L. Lingappan, S. Ravi, and N. K. Jha. Test generation for non-separable RTL controller-datapath circuits using a satisfiability based approach. In *International Conference on Computer Design*, pages 187–193, 2003.

[53] S. Manich, A. Gabarró, M. Lopez, J. Figueras, P. Girard, L. Guiller, C. Landrault, S. Pravossoudovitch, P. Teixeira, and M. Santos. Low power BIST by filtering non-detecting vectors. *Journal of Electronic Testing: Theory and Applications*, 16(3):193–202, 2000.

[54] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., 1992.

[55] P. C. Maxwell, R. C. Aitken, V. Johansen, and I. Chiang. The effect of different test sets on quality level prediction: When is 80% better than 90%? In *International Test Conference*, pages 358–364, 1991.

[56] E. J. McCluskey and C. W. Tseng. Stuck-fault tests vs. actual defects. In *International Test Conference*, pages 336–343, 2000.

[57] K. L. McMillan. *Symbolic model checking*. PhD thesis, Carnegie Mellon University, 1992.

[58] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1-3):279–309, 2000.

[59] P. Mishra and N. Dutt. Automatic functional test program generation for pipelined processors using model checking. In *High Level Design Validation and Test Workshop*, pages 99–103, 2002.

[60] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535, 2001.

[61] B. T. Murray and J. P. Hayes. Hierarchical test generation using precomputed tests for modules. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):594–603, 1990.

[62] T. M. Niermann, W. T. Cheng, and J. H. Patel. PROOFS: a fast, memory-efficient sequential circuit fault simulator. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 11(2):198–207, 1992.

[63] T. M. Niermann and J. H. Patel. HITEC: A test generation package for sequential circuits. In *European Design automation conference*, pages 214–218, 1991.

[64] C. A. Papachristou, F. Martin, and M. Nourani. Microprocessor based testing for core-based system on chip. In *Design automation conference*, pages 586–591, 1999.

[65] P. Parvathala, K. Maneparambil, and W. Lindsay. FRITS - a microprocessor functional BIST method. In *International Test Conference*, pages 590–598, 2002.

[66] J. Rearick. Too much delay fault coverage is a bad thing. In *International Test Conference*, pages 624–633, 2001.

[67] J. P. Roth. Diagnosis of automata failures: A calculus and a method. *IBM Journal of Research and Development*, 10(4):278–291, 1966.

[68] K. Roy and J. A. Abraham. High level test generation using data flow descriptions. In *European Design Automation Conference*, pages 480–484, 1990.

[69] J. Savir and S. Patil. Broad-side delay test. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1057–1064, 1994.

[70] M. H. Schulz and E. Auth. ESSENTIAL: An efficient self-learning test pattern generation algorithm for sequential circuits. In *International Test Conference*, pages 28–37, 1989.

[71] F. F. Sellers, M. Y. Hsiao, and L. W. Bearnson. Analyzing errors with the boolean difference. *IEEE Transactions on Computers*, 17(7):676–683, 1968.

[72] J. Shen and J. A. Abraham. Native mode functional test generation for processors with applications to self test and design validation. In *International Test Conference*, pages 990–999, 1998.

[73] L. Shen and S. Y. H. Su. A functional testing method for microprocessors. *IEEE Transactions on Computers*, 37(10):1288–1293, 1988.

[74] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara. Instruction-based delay fault self-testing of pipelined processor cores. In *International Symposium on Circuits and Systems*, pages 5686–5689, 2005.

[75] G. L. Smith. Model for delay faults based upon paths. In *International Test Conference*, pages 342–349, 1985.

[76] M. H. Tehranipour, M. Nourani, S. M. Fakhraie, and A. Afzali-Kusha. Systematic test program generation for SOC testing using embedded processor. In *International Symposium on Circuits and Systems*, pages 541 – 544, 2003.

[77] S. M. Thatte and J. A. Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, 29(6):429–441, 1980.

[78] N. A. Touba and B. Pouya. Testing embedded cores using partial isolation rings. In *VLSI test symposium*, pages 10–16, 1997.

[79] R. S. Tupuri and J. A. Abraham. A novel functional test generation method for processors using commercial ATPG. In *International Test Conference*, pages 743–752, 1997.

[80] R. S. Tupuri and J. A. Abraham. A novel hierarchical test generation method for processors. In *International Conference on VLSI Design*, pages 540–541, 1997.

[81] R. S. Tupuri, J. A. Abraham, and D. G. Saab. Hierarchical test generation for systems on a chip. In *VLSI Design Conference*, pages 198–203, 2000.

[82] V. M. Vedula, J. A. Abraham, J. Bhadra, and R. S. Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing: Theory and Applications*, 19(2):149–160, 2003.

[83] A. Virazel, R. David, P. Girard, C. Landrault, and S. Pravossoudovitch. Delay fault testing: Choosing between random SIC and random MIC test sequences. *Journal of Electronic Testing: Theory and Applications*, 17(3-4):233–241, 2001.

[84] P. Vishakantaiah, J. Abraham, and M. Abadir. Automatic test knowledge extraction from VHDL (ATKET). In *Design Automation Conference*, pages 273–278, 1992.

[85] V. Viswanath, J. A. Abraham, and W. A. Hunt Jr. Automatic insertion of low power annotations in RTL for pipelined microprocessors. In *Conference on Design Automation and Test in Europe*, pages 496–501, 2006.

[86] J. A. Waicukauski, E. Lindbloom, B. K. Rosen, and V. S. Iyengar. Transition fault simulation. *IEEE Design and Test of computers*, 4(2):32–38, 1987.

[87] L. Whetsel. An IEEE 1149.1-based test access architecture for ICs with embedded cores. In *International Test Conference*, pages 69–78, 1997.

[88] F. Xin, M. Ciesielski, and I. G. Harris. Design validation of behavioral VHDL descriptions for arbitrary fault models. In *European Test Symposium*, pages 156–161, 2005.

[89] OR1200 RISC processor. http://www.opencores.org.

[90] L. Zhang, I. Ghosh, and M. Hsiao. Efficient sequential ATPG for functional RTL circuits. In *International Test Conference*, pages 290–298, 2003.

[91] Y. Zorian. A distributed BIST control scheme for complex VLSI devices. In *VLSI Test Symposium*, pages 4–9, 1993.

[92] Y. Zorian. Test requirements for embedded core-based systems and IEEE P1500. In *International Test conference*, pages 191–199, 1997.

Vita

Sankaranarayanan Gurumurthy received the Bachelor of Technology degree in Electronics and Communications Engineering from the Regional Engineering College (now known as National Institute of Technology), Warangal, India in 2001. He joined the graduate program of University of Colorado at Boulder in August 2001 to pursue a Masters degree. He completed the program in 2003 with a thesis in formal verification of hardware systems and joined Obsidian Software in Austin for an internship. He joined the University of Texas at Austin in the spring of 2004 for Doctoral studies in Electrical and Computer Engineering. His area of research has been functional testing of processors and systems-on-a-chip during since then. He has also interned with Intel in Hillsboro, Oregon and Austin.

Permanent address: No. 16, V. K. Nagar,
Nadukuttagai, Tiruninravur,
Tamilnadu, India 602024

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.